

# 第 1 章 控制台五子棋

## 1.1 引言

控制台五子棋，顾名思义，就是在 Java 控制台运行的五子棋游戏，需要用户用键盘输入棋子的位置来进行游戏。

由于是在控制台下面运行的程序，所以并没有漂亮的游戏界面，与及鼠标操作等东西，只是在一片黑色控制台环境下进行游戏，游戏的可玩性并不高，似乎这并不是一个完整的游戏。虽然如此，但事实上，一个程序最重要的并不是界面，而是处理各种业务逻辑与数据的方法，只要掌握了核心的方法，掌握基础的知识，便更容易学习 `awt,swing` 等图形用户界面的编写，万变不离其宗，写起有操作界面的程序也会变得更加容易，更加随心应手。而本章的主要目的让读者掌握与理解 Java 编程的基础知识，因此，掌握本章五子棋的实现原理，对于学习以后的章节将会非常有帮助。作为本书的第一章内容，我们在本章中将使用最简单的方式来实现一个控制台五子棋游戏。

### 1.1.1 五子棋介绍

五子棋是起源于中国古代的传统黑白棋种之一。现代五子棋日文称之为“连珠”，英译为“Renju”，英文称之为“Gobang”或“FIR”（Five in a Row 的缩写），亦有“连五子”、“五子连”、“串珠”、“五目”、“五目碰”、“五格”等多种称谓。五子棋游戏是一个比较大众的棋类游戏，大多数人都会玩这个游戏，五子棋的玩法与规则如下：

- ❑ 五子棋是两个人之间进行的竞技活动，由于对黑方白方规则不同，黑棋必须先行（本章节设计的游戏，黑棋与白棋的规则一样，但一样由黑棋先下）。
- ❑ 五子棋专用盘为 **15x15**，五连子的方向为横、竖、斜。
- ❑ 在棋盘上以对局双方均不可能形成五连为和棋。
- ❑ 首先形成五连子的一方为赢。

五子棋必须由双方进行游戏，当某一方按照一定规则连成五个棋子的时候，该游戏方就胜利，在本章中，我们并不需要做到对战形式的，我们可以设计一个简单的“电脑”来做我们的对手，当我们下完棋后，这个简单的“电脑”就随便在棋盘中下一个棋，当然，如果想做更强大的“电脑”我们可以编写程序来实现，当我们下棋的时候，这个“电脑”就对我们所下的棋子进行检测，并将棋子下到最恰当的位置。本章主要目的是展现五子棋的实现原理，如果读者有兴趣，可以自行开发强大的“人工智能电脑”来进行游戏。

### 1.1.2 输入输出约定

玩家必须以 (x,y) 的方式输入棋盘的坐标，其中，x 代表棋坐标，y 代表竖坐标。x 与 y 的值必须是 1 到 N（棋盘的大小）的正数。

系统询问玩家是否继续游戏时，玩家输入 y 是代表继续，输入其它则代表退出游戏。

“●”代表黑子，“○”代表白子。当玩家以 (x,y) 的形式输入下棋的坐标后，游戏中就可以根据玩家所下的坐标，再去将棋子放置到棋盘中。我们可以将棋盘看作一个二维数组，填充着棋盘式的标志（“十”），玩家下棋后，将棋子替换原来的标志，最后再执行输入。由于本章是在控制台中进行打印，因此只需要使用 `System.out.println` 来进行打印即可，如果需要有界面的五子棋游戏，例如使用

swing 或者 awt，可以使用相应的方法，将二维数组“画”到界面中。因此，不管是使用 swing、awt 或者其他界面技术，五子棋的实现原理几乎大同小异。

## 1.2 了解游戏流程描述

在开发五子棋之前，我们先了解一下游戏的整个游戏流程，了解游戏的流程，有助于我们在开发的过程中可以清晰的掌握程序结构，对于实现功能有莫大的帮助，五子棋的具体流程如图 1.1 所示。

五子棋游戏流程

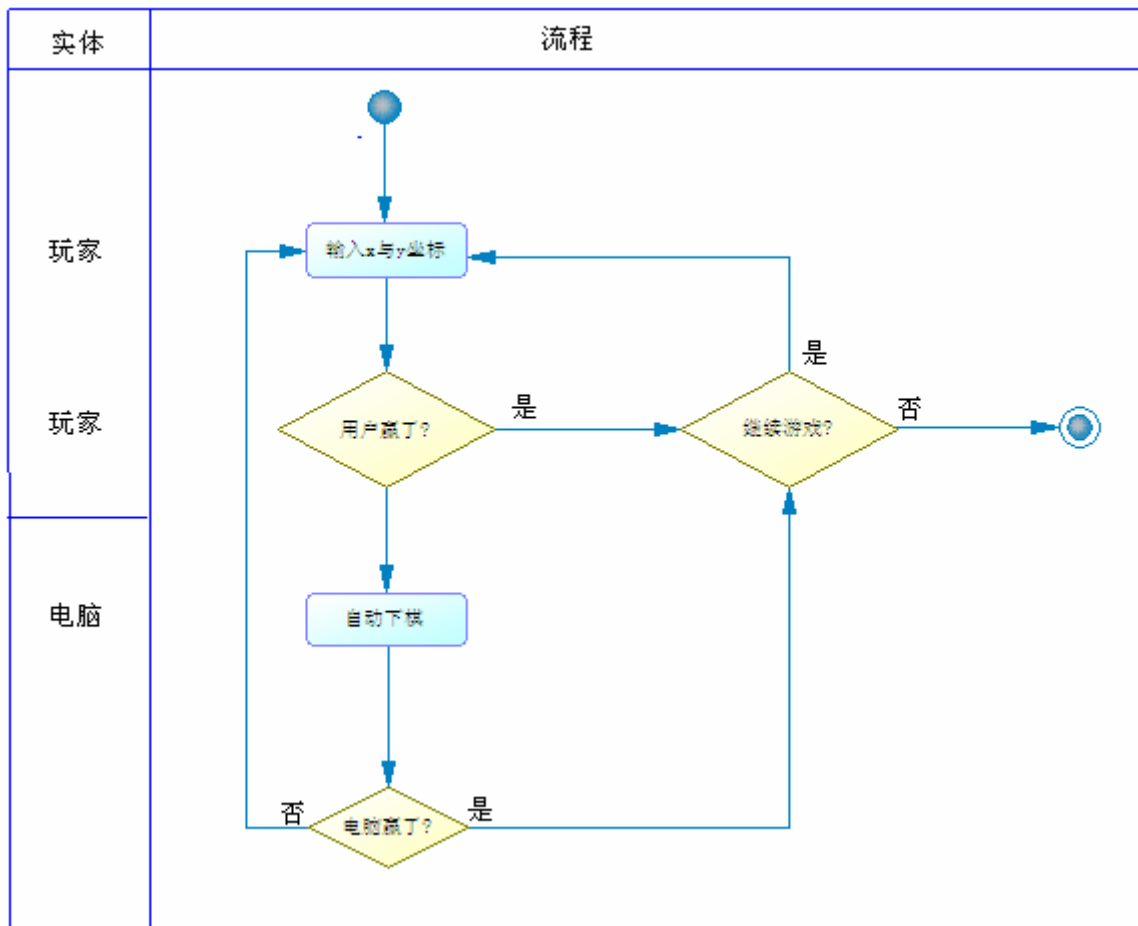


图 1.1 五子棋游戏流程

### 1.2.1 玩家输入坐标

游戏开始，系统在控制台中打印出棋盘，玩家根据这个棋盘，选定下棋的位置坐标后，在控制台中输入相应的坐标，系统读取玩家所输入的坐标并进行相应的分析，如果玩家所下的棋使得玩家游戏胜利，则系统询问是否继续游戏。

系统读取了玩家输入的坐标后，除了判断游戏是否胜利外，还需要判断玩家输入的坐标中是否已经存在了相应的棋子，如果存在的话，需要再次提示玩家，重新输入。

### 1.2.2 “电脑”下棋

玩家输入了坐标，系统判断玩家没有游戏胜利后，就应该轮到“电脑”下棋，在本章的开头中我们已经讲到，本章可以实现一个简单的电脑来进行游戏，只需要随便的产生棋盘坐标，就可以让“电脑”在相应的坐标中下棋。如果电脑随机产生的坐标中已经存在棋子，那么我们可以重新随机产生坐标，直到产生的坐标上没有存在棋子为止。当“电脑”下完棋后，就可以使用同样的判断方式（判断是否五子相连）来判断“电脑”所下的棋子是否已经使得游戏胜利，如果游戏胜利，同样地去提示玩家，电脑已经胜利了。

在本章我们并不需要实现强大的人工智能“电脑”，只需要简单的随机产生坐标即可。

## 1.3 创建游戏的各个对象

这里设计三个类来完成游戏的功能，棋盘类（Chessboard），游戏类（GobangGame）与棋子类（Chessman）（枚举类），类的关系如图 1.2 所示，从图中可以看出，Chessboard 依赖于 GobangGame，gobangGame 的改变，会影响到 Chessboard 状态的改变，而 Chessman 与 GobangGame 是一个聚合关系。下面一一介绍。

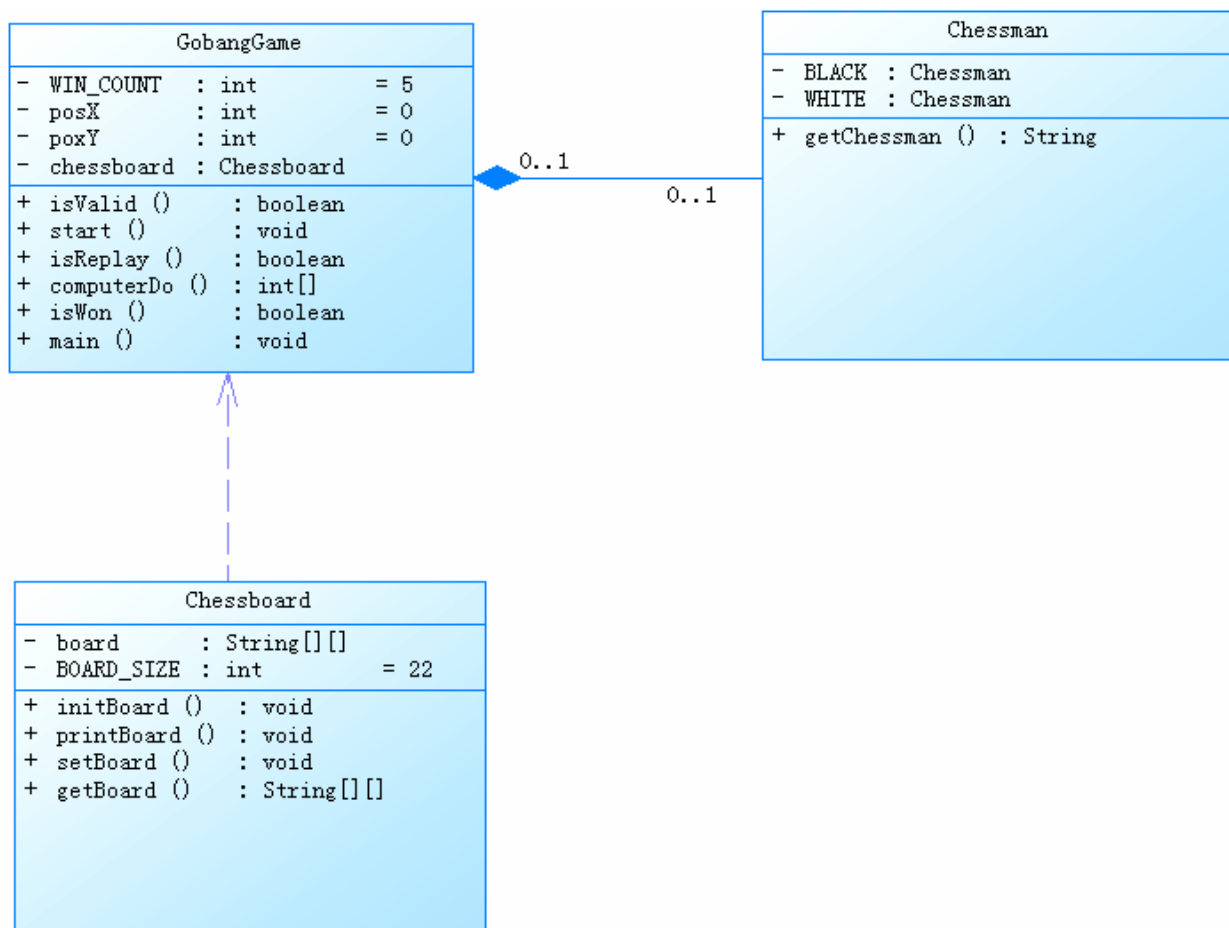


图 1.2 五子棋类图

### 1.3.1 Chessboard类

要进行五子棋游戏，必须有有一个棋盘，而这个类主要控制棋盘的初始化，输出与及增加新的棋子。这个类包含以下方法：

- ❑ `void initBoard()`，这个方法用于初始化棋盘，开始新的游戏时，应该调用此方法，初始化出一个新的空棋盘。
- ❑ `void printBoard()`，此方法用于在控制台输出棋盘，各方每一完一颗棋子后，由于棋盘上棋子的状态有改变，所以必须调用此方法重新输入棋盘。
- ❑ `void setBoard( int posX , int posY , String chessman )`，`posX` 与 `posY` 是新下棋子的 `x` 与 `y` 坐标，`chessman` 是新下棋子的类型（黑子与白子），每下完一颗棋子后，通过调用此方法把棋子设置到棋盘上。
- ❑ `String[][] getBoard()`，返回棋盘，返回类型是保存棋盘的二维数组。

当我们需要初始化棋盘的时候，可以直接调用 `Chessboard` 的 `initBoard` 方法，我们需要考虑该方法需要实现的功能：初始化棋盘。由于我们将棋盘看作是一个二维数组，因此 `initBoard` 就需要帮我们去创建一个二维数组，创建二维数组可以使用以下代码。

代码清单：code\gobang\src\org\crazyit\gobang\Chessboard.java

```
Object[][] array = new Object[size][size];
for (int i = 0; i < array.length; i++) {
    for (int j = 0; j < array[i].length; j++) {
        array[i][j] = new Object();
    }
}
```

以上代码创建一个固定大小（一维与二维大小）的二维数组，再通过嵌套循环为数组中的每一个元素进行赋值。在游戏中如果我们进行了下棋的操作，可以直接改变这个数组的某一个元素值。在创建 `Chessboard` 类时，我们就需要发挥面向对象的思维，在我们的程序中，所有看到的或者想的事物，我们都可以将其抽象成具体的某个对象，并赋予一定的属性与行为。在设计对象的过程中，如果有某些事物拿捏不准，不知如何设计属性或者行为，可以将其设计成接口或者抽象类。

`Chessboard` 中提供了一个 `printBoard` 的方法用于打印棋盘，在本章中，我们就需要将棋盘数组打印到控制台中，因此该方法可以简单的调用 `System.out.print` 去打印相关的字符串。需要注意的是，由于 `printBoard` 方法是没有参数的，因此我们需要为 `Chessboard` 提供一个二维数组变量，当调用 `printBoard` 方法的时候，将对象内的二维数组打印，我们可以将 `Chessboard` 看作一个有状态的 Java 对象，有状态的 Java 对象可以理解成一个 Java 对象保存一些与该对象相关的状态属性，如果该对象没有保存与该对象相关的状态属性，那么我们可以将这个对象看成一个无状态的 Java 对象。

当外部调用 `Chessboard` 的 `setBoard` 方法时，就可以将某个值设置到 `Chessboard` 中的二维数组里，告诉 `Chessboard` 玩家或者“电脑”在某个位置下了怎样的棋子。

### 1.3.2 Chessman类

`Chessman` 类是一个枚举类，此类是构造器私有的，不能直接创建，里面有 `BLACK` 与 `WHITE` 两个静态属性，代表黑子与白子枚举类，两个表态属性都是 `Chessman` 类型的，要获取棋子，则通过这两个属性调用以下的方法获取棋子：

- ❑ `String getChessman()`，返回 `String` 类型的棋子实例，“●”或者“○”。
- 如果我们需要得到棋子的字符串（“●”或者“○”），可以使用以下的代码。

代码清单：code\gobang\src\org\crazyit\gobang\Chessman.java

```
Chessman.BLACK.getChessman();
```



### 1.3.3 GobangGame类

GobangGame 类是进行游戏的类，Chessboard 依赖于此类，此类控制游戏的开始，重玩与结束，并影响 Chessboard 类。主要包含以下构造器与方法：

- ❑ GobangGame(), 默认无参数构造器。
- ❑ GobangGame( Chessboard chessboard ), 有参数构造器，以一个 Chessboard 实例去初始化这个类。
- ❑ boolean isValid( String inputStr ), 此方法验证控制台的输入字符串是否合法，如果合法，返回 true，如果不合法，则返回 false，此方法抛出 Exception 异常。
- ❑ void start(), 开始游戏。此方法抛出 Exception 异常。
- ❑ boolean isReplay( String chessman ), 是否重新开始游戏，如果是，返回 true，否则返回 false，参数 chessman 代表黑子或白子。
- ❑ int[] computerDo(), 计算机随机下棋，由计算机自动设置棋盘，并返回包含新下棋子位置 x 与 y 坐标的 int[] 数组类型。
- ❑ boolean isWon( int posX, int posY, String ico ), 判断输赢，参数 posX 与 posY 代表新下棋子的 x 与 y 坐标，ico 代表新下的棋子类型，如果赢了，返回 true，否则返回 false。

GobangGame 是我们五子棋游戏的主体类，游戏里面所有的处理都在该类中实现。GobangGame 中的 isValid 方法用于验证控制台的输入，玩家主要在控制台输入下棋的坐标，下棋的坐标的字符串形式为:x,y,我们需要对字符串进行处理得到x和y的值,如果玩家输入的字符串不符合系统要求,则isValid方法返回 false,只有当该方法返回 true 的时候,才会去修改 Chessboard 的二维数组。

GobangGame 中提供了一个 start 方法，用于游戏的开始，我们需要考虑游戏开始的行为，例如需要初始化棋盘（调用 Chessboard 的 init 方法），需要开始从控制台读取玩家的输入信息、打印棋盘，验证控制台输入的信息等，这些功能我们将在下面的章节中加以描述。

当轮到“电脑”下棋的时候，我们需要随机生成电脑的下棋坐标，GobangGame 中的 computerDo 方法用于随机产生坐标。

判断一局游戏是否胜利，可以调用 GobangGame 的 isWon 方法，该方法判断游戏是否胜利，是五子棋中最主要的方法，五子棋是否可以相连的所有逻辑，都会在该方法中实现。isWon 方法会在每次下棋后（玩家下棋或者“电脑”下棋）调用。

到此，游戏中的三个对象已经设计完成，这三个对象中已经定义好了各种方法，并在前面章节中详细描述了各个方法的作用，在下面章节中我们将开始对这三个对象所定义的方法进行实现。当然，如果需要做到更好的程序解耦，我们可以使用一些设计模式，例如将游戏规则写成一个具体的算法，可以使用策略模式，如果需要产生出不同的棋子（将控制台换成其他界面），可以编写棋子工厂等。但是本章主要目的是展现一个最简单的五子棋，因此本章中并不涉及任何具体的设计模式。

## 1.4 棋盘类实现

在此类中，主要是用一个 String[][] 类型的二维数组 board 去保存棋盘，board [i][j] 代表棋盘的某个位置(i 代表 x 坐标，j 代表 y 坐标)，如果此位置没有棋子，默认值为“十”，如果有棋子，board [i][j] 的值为“●”或者“○”。用一个不可改变的常量 BOARD\_SIZE 来表示棋盘的大小，所以保存这个棋盘的是一个 BOARD\_SIZE\*BOARD\_SIZE 的二维数组。图 1.3 描述了为什么需要使用一个二维数组来代表一个棋盘，如果把棋盘的一列当做一个数组，那么 N 列的棋盘就是一个二维数组，用数组能很好的存储与表现这个棋盘。

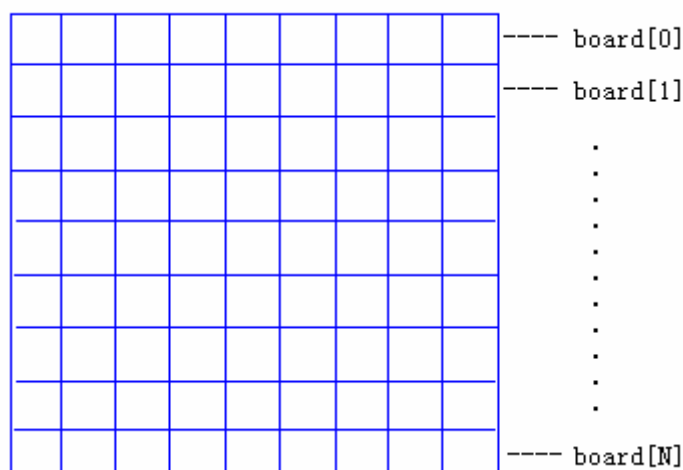


图 1.3 棋盘与数组的关系

### 1.4.1 初始化棋盘

在 1.3 节介绍过，此类主要是实现棋盘初始化、输出、与更新，在这节使用代码一步一步地实现各个功能。首先我们需要初始化棋盘的实现，看以下代码片段。

代码清单：code\gobang\src\org\crazyit\gobang\Chessboard.java

```
public void initBoard() {  
    //初始化棋盘数组  
    board = new String[BOARD_SIZE][BOARD_SIZE];  
    //把每个元素赋值为“十”，用于控制台输出棋盘  
    for( int i = 0 ; i < BOARD_SIZE ; i++ ) {  
        for( int j = 0 ; j < BOARD_SIZE ; j++ ) {  
            board[i][j] = "十";  
        }  
    }  
}
```

上面代码中，`BOARD_SIZE` 是代表棋盘的大小，用一个 `String[][]` 类型的二维数组来代表棋盘，创建此数组后，通过迭代为每个数组元素的值赋为“十”来初始化棋盘。创建了棋盘数组后，如果需要定位到棋盘的某个位置，只需要得到棋盘数组的一维值与二维值即可，例如处理玩家下棋动作的时候，可以将数组中具体的某个“十”替换成具体的棋子字符串。

### 1.4.2 输出棋盘

输出棋盘，只需要 `Chessboard` 的 `board` 属性（二维数组）的每一个值，打印到控制台中。如果可以做到更好的扩展性，我们可以在二维数组中存放棋子对象，而不是简单的字符串，那么存放在二维数组中的每一个棋子对象，都可以实现某个棋子接口或者继承棋子的抽象类，这样可以更好的做到游戏的扩展性。当然，我们在本章为了简单起见，只在该二维数组中存放需要打印的字符串，打印时只需要得到具体的某个二维数组的元素，将其打印即可。

代码清单：code\gobang\src\org\crazyit\gobang\Chessboard.java

```
public void printBoard() {  
    //打印每个数组元素  
    for( int i = 0 ; i < BOARD_SIZE ; i++ ) {
```

```
for( int j = 0 ; j < BOARD_SIZE ; j++ ) {  
    //打印后不换行  
    System.out.print( board[i][j] );  
}  
//每打印完一行数组元素就换行一次  
System.out.print("\n");  
}  
}
```

棋盘的输出与棋盘的初始化相类似，都是要遍历保存棋盘的数组，只不过是每遍历到一个元素都要输出来，注意到这里的输出方法用的是 `System.out.print()` 而不是常用的 `System.out.println()`，这里因为 `System.out.println()` 方法是输出后自动换行的，如果使用此方法，便达不到我们需要的效果，棋盘的输出效果如图 1.4。

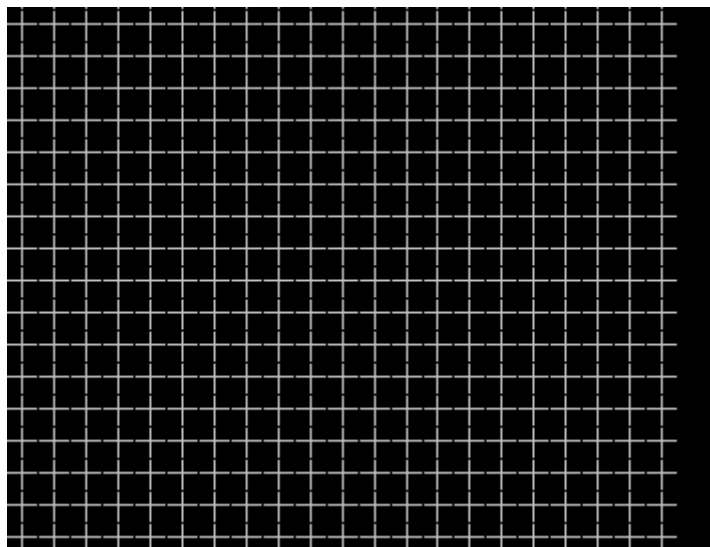


图 1.4 控制台五子棋的棋盘

打印出来的效果，就好像在控制台中出现了一个棋盘。

### 1.4.3 获取棋盘

在 `Chessboard` 中提供了一个 `getBoard` 的方法，用于返回本对象的棋盘二维数组，该方法一般在游戏类 `GobangGame` 中调用，游戏类得到棋盘的二维数组，可以用于判断棋盘中的某一个位置是否有棋子或者计算游戏是否胜利。

`getBoard` 方法只需要将本对象中的 `board`（二维数组）返回即可，代码如下。

代码清单：code\gobang\src\org\crazyit\gobang\Chessboard.java

```
/**  
 * 返回棋盘  
 * @return 返回棋盘  
 */  
public String[][] getBoard() {  
    return this.board;  
}
```

到此，棋盘类的几个方法都已经实现，该类的主要功能是创建棋盘、打印棋盘等，实现的过程中涉及了一些 `Java` 语言的基本操作，例如嵌套循环、创建二维数组等。在下面的小节中，我们将去实现游戏的核心部分。

## 1.5 棋子枚举类实现

在某些情况下，一个类的属性是有限而且固定的（在某些情况下），例如本章中的棋子类，它只有两个对象，黑棋和白棋。这种实例有限而且固定的类，在 Java 里面称为枚举类，枚举类的关键字用 `enum` 而不是 `class`，此类中有两个枚举属性 `BLACK` 和 `WHITE`，代表黑子与白子，代码实现如下：

代码清单：code\gobang\src\org\crazyit\gobang\Chessman.java

```
public enum Chessman {  
    BLACK("●"),WHITE("○");  
    private String chessman;  
    /**  
     * 私有构造器  
     */  
    private Chessman(String chessman) {  
        this.chessman = chessman;  
    }  
    /**  
     * @return 黑棋或者白棋  
     */  
    public String getChessman() {  
        return this.chessman;  
    }  
}
```

在上面的代码中，可以看到，枚举类是用 `enum` 关键字代替了 `class` 关键字，看到此枚举类的构造器的权限修饰符是 `private`，也是表明此类是不可以通过外部创建的，只能在此类的内部创建，这是为了保证此对象只有黑子与白子两种类型。黑体代码是列出枚举值，实际上就是调用私有构造器创建此对象，等同以下代码：

```
public static final Chessman BLACK = new Chessman("●");  
public static final Chessman WHITE = new Chessman("○");
```

由于 `BLACK` 与 `WHITE` 两个属性是静态的，所以要获取黑子或者白子，可以通过以下代码来获得：

```
Chessman.BLACK.getChessman();  
Chessman.WHITE.getChessman();
```

在控制台中，我们可以使用这种方式来确定棋子的字符串，如果我们需要在 `swing` 或者其他界面中展示一个棋子，可能需要为具体的某个棋子保存相应的棋子图片，在本章中，由于棋子只是普通的两个字符串，因此可以直接写成枚举对象即可。

如果你希望你的程能有更好的扩展性，笔者建议可以根据情况建立棋子接口，并提供白棋与黑棋的实现类，我们在棋盘二维数组中存放的只是某个接口，而不是具体的类，这样，提高了程序的可扩展性，在本小节的开头提到：在某些情况下，一个类中的属性有限并且是固定的。但是在我们开发的实际情况中（特别是做企业应用），随着业务的不断变化，类的不可变几乎是不可能的。举个例子，如果需要将本章中的五子棋迁移到 `swing` 界面中，那么该棋子枚举类就不得不更改了。

虽然本章是为了做一个较为简单的五子棋，但更多的想向大家展现面向对象的思维。

## 1.6 游戏类实现

本章中的游戏类是 `GogangGame`，在该类中，主要控制游戏的开始，重新开始与结束，验证玩家输入的合法性，判断游戏的输与赢，调用棋盘类来初始化棋盘，打印棋盘，使用棋子类去设置棋盘等。此类中有四个属性，两个 `int` 类型的 `posX` 与 `poxY`，用来存储玩家现在输入的 `x` 与 `y` 坐标（`x` 和 `y` 坐标

是指玩家输入的数字对应棋盘数组中的一维值与二维值), 一个默认值为5的int类型常量WIN\_COUNT, 游戏胜利需要连接的棋子达到的连子数目, 由于是五子棋, 因此只需要5个棋子相连, 游戏就胜利。还有一个Chessboard类型的变量chessboard, 用来表示棋盘, 游戏中就只用到一个棋盘, 该对象可以使用初始化棋盘、打印棋盘、获得棋盘(数组)等方法。

### 1.6.1 使用BufferedReader获取键盘输入

BufferedReader 是 Java IO 流中的一个字符包装流, 它必须建立在字符流的基础之上。该对象可以从输入流中读取文本, 但标准输入: System.in 是字节流, 所以程序需要使用转换流 InputStreamReader 将其包装成字符流。所以程序中用于获取键盘的输入采用以下代码创建。

```
//获取键盘的输入
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String inputStr = null;
//br.readLine:每当键盘输入一行内容按回车键, 则输入的内容被 br 读取到
while( (inputStr = br.readLine()) != null ) {
    /**
     * 处理键盘输入
     */
}
```

BufferedReader 中有一个 readLine()方法, 此方法总是读取下一行的输入字符串, 如果没有下一行, 则返回 null。当得到玩家输入的字符后, 我们可以进这些字符进行验证, 验证完后, 如果字符串符合系统要求, 可以在验证处使用 continue 跳出本次循环。

如果需要读取输入, 我们就需要为这些输入作出不同的判断, 例如, 玩家输入了 y (继续游戏), 那么我们就需要判断玩家输入了 y 后程序所需要执行哪些操作, 因此, 这样会为 while 循环体中增加许多的 if 语句, 这些 if 语句会影响程序的可读性, 如果要将这些 if 语句去掉, 我们可以将每个 if 中的代码抽取出来, 作为具体的一个处理类。这样做不仅减少 while 循环体中的代码, 而且可以使得程序更加清晰, 程序的耦合度更低, while 循环体中只负责读取玩家输入的字符串, 而具体的处理则不必由该方法来执行。由于本章中的代码与动作相对较少, 因此并不涉及如何实现以上所说的处理模式, 更深入的可以查看“仿 QQ 游戏大厅”一章。

### 1.6.2 验证玩家输入字符串的合法性

根据引言中提到的输入约定, 玩家在控制台输入的字符串必须是以(x,y)的方式输入, 还需要验证输入的字符串是否能转换为数字, 是否超越棋盘的边界(小于等于1, 大于等于棋盘数组的长度), 并且需要判断该位置是否已经存在棋子, 具体判断流程如图1.5所示。

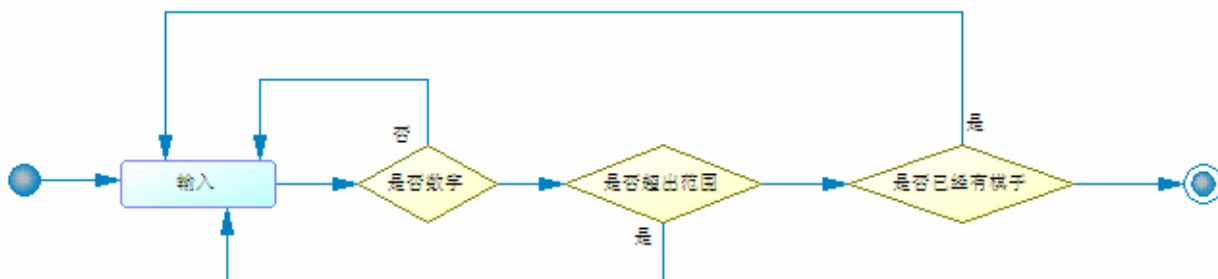


图 1.5 验证流程

首先, x 与 y 必须是一个数字, 由以下代码验证。

代码清单: code\gobang\src\org\crazyit\gobang\GobangGame.java

```
//将用户输入的字符串以逗号(,)作为分隔, 分隔成两个字符串
String[] posStrArr = inputStr.split(",");
try {
    posX = Integer.parseInt( posStrArr[0] ) - 1;
    posY = Integer.parseInt( posStrArr[1] ) - 1;
} catch (NumberFormatException e) {
    chessboard.printBoard();
    System.out.println("请以(数字,数字)的格式输入: ");
    return false;
}
```

当我们调用 `Integer.parseInt` 方法将字符串转换成一个 `Integer` 类型的时候, 如果需要转换的字符串不能转换成某一个数字, 该方法将会抛出 `NumberFormatException` 异常, 我们可以使用 `catch` 将该异常捕获, 提示玩家需要重新输入棋子坐标。除了判断玩家输入的字符串是否符合我们游戏所要求的格式外, 还需要判断玩家输入的坐标范围, 即该范围不可小于 1 并不可大于棋盘数组的最大值, 例如棋盘是 10 乘 10, 但玩家输入了 11, 那么将会抛出 `ArrayIndexOutOfBoundsException` 异常, 因此, `x` 与 `y` 的范围只能是大于 1 与小于 `N` (棋盘的大小), 如果超出这个范围, 则需要提示玩家重新输入, 由以下代码验证。

代码清单: `code\gobang\src\org\crazyit\gobang\GobangGame.java`

```
//检查输入数值是否在范围之内
if( posX < 0 || posX >= Chessboard.BOARD_SIZE || posY < 0
    || posY >= Chessboard.BOARD_SIZE ) {
    chessboard.printBoard();
    System.out.println( "X 与 Y 坐标只能大于等于 1,与小于等于"
        + Chessboard.BOARD_SIZE + ",请重新输入: " );
    return false;
}
```

验证了输入坐标的合法性后, 还需要验证玩家输入的坐标中是否已经存在棋子, 我们通过 `Chessboard` 对象中的 `getBoard` 方法可以得到棋盘的二维数组, 根据玩家输入的坐标得到数组中的元素, 再判断元素是否已经是一个棋子 (“●” 或者 “○”), 如果该坐标中已经有棋子 (元素值为 “十”), 则提示玩家重新输入, 由以下代码验证。

代码清单: `code\gobang\src\org\crazyit\gobang\GobangGame.java`

```
//检查输入的位置是否已经有棋子
String[][] board = chessboard.getBoard();
if( board[posX][posY] != "十" ) {
    chessboard.printBoard();
    System.out.println( "此位置已经有棋子, 请重新输入: " );
    return false;
}
```

以上代码中, 如果 `board[i][j]` 不等于 “十” (“十” 是棋盘每个位置的默认值), 则证明此位置有棋子, 需要提示玩家重新输入。这里需要注意的是, 如果没有前一个判断 (判断输入的坐标是否超过棋盘范围), 那么通过棋盘数组获取某个元素时, 就会抛出 `ArrayIndexOutOfBoundsException` 异常。

### 1.6.3 判断输赢

判断游戏输赢, 需要在玩家输入了坐标并通过了合法性验证后 (输入的坐标), 再执行输赢的验证, 同样地, 如果是 “电脑” 随机生成的坐标, 我们同样的需要进行输赢验证, 因此, 我们已经将判断输赢的行为, 独立成一个 `isWon` 方法 (详细请看 1.3.3 中的 `GobangGame` 类)。

判断输赢在本章的程序中稍微复杂, 有两种方法来判断输赢:

- ❑ 每次下完一颗棋子，就通过程序从横、竖、斜各个方向扫描棋盘，如果在某个方向中，有同种颜色的棋子达到五连子，则此颜色的玩家为赢。如果没有相同颜色的棋子达到五连子，则继续游戏。该判断方法需要遍历整个棋盘，也就是意味着每次下棋后（玩家或者“电脑”）都需要对棋盘进行遍历，这样对程序的性能会造成一定的影响。
- ❑ 每次下完一颗棋子，以该棋子为中心，扫描在此棋子所在范围内的横、竖、斜方向，验证加上此棋子有没有形成五连子，如果形成五连子，则下棋子的玩家为赢。此方法与前面的方法比较，因为不需要扫描整个棋盘，所以更加快速，本程序使用的是此方法，该方法的原理如图 1.6 所示。

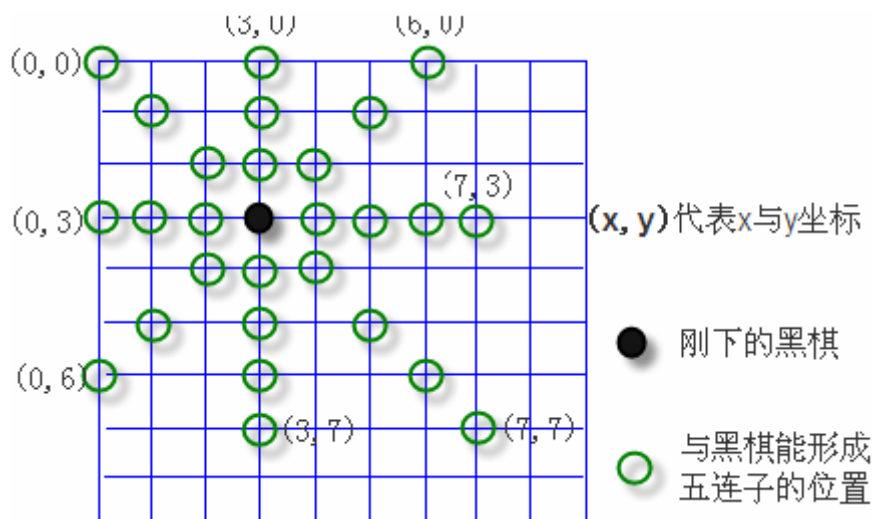


图 1.6 五连子

在图 1.6 中可以看出，(0, 0)，(0, 3)，(0, 6)，(3, 0)，(6, 0)，(3, 7)，(7, 3)，(7, 7) 这些坐标都是此黑棋能形成五连子的最小或者最大位置的棋子，如果各个方向有足够的空间，就延伸到第五颗棋子，如果没有，就只延伸到边界。所以，只要能计算出任意一颗棋子的这些位置，我们就可以判断游戏的输赢，并且是以该棋子为中心向周围进行遍历。以下是判断输赢的代码实现。

代码清单：code\gobang\src\org\crazyit\gobang\GobangGame.java

```
//直线起点的 X 坐标
int startX = 0;
//直线起点 Y 坐标
int startY = 0;
//直线结束 X 坐标
int endX = Chessboard.BOARD_SIZE - 1;
//直线结束 Y 坐标
int endY = endX;
//同条直线上相邻棋子累积数
int sameCount = 0;
int temp = 0;
//计算起点的最小 X 坐标与 Y 坐标
temp = posX - WIN_COUNT + 1;
startX = temp < 0 ? 0 : temp;
temp = posY - WIN_COUNT + 1;
startY = temp < 0 ? 0 : temp;
//计算终点的最大 X 坐标与 Y 坐标
temp = posX + WIN_COUNT - 1;
```

```

endX = temp > Chessboard.BOARD_SIZE - 1 ?
    Chessboard.BOARD_SIZE - 1 : temp;
temp = posY + WIN_COUNT - 1;
endY = temp > Chessboard.BOARD_SIZE - 1 ?
    Chessboard.BOARD_SIZE - 1 : temp;
//从左到右方向计算相同相邻棋子的数目
String[][] board = chessboard.getBoard();
for( int i = startY; i < endY; i++) {
    if( board[posX][i] == ico && board[posX][i+1] == ico ) {
        sameCount++;
    } else if( sameCount != WIN_COUNT - 1 ) {
        sameCount = 0;
    }
}
}

```

从上面代码中可以看到，首先是计算出在这颗棋子的直线上(横、竖、斜方向)能达到五连子的最小 x、y 坐标与最大 x、y 坐标，然后从最小 x、y 坐标访问到最大 x、y 坐标，如果此颜色棋子的相连累积数目达到五连子，则为赢。以上代码只是实现横向遍历判断，竖向遍历与斜向遍历的判断方法与横向遍历的实现基本类似。这里需要注意的是，当遇到一个可以相边的棋子，就需要为 `sameCount` 值加 1。

#### 1.6.4 计算机随机下棋

我们在前面章节中说到，使用一个简单的方式来产生一个“电脑”与我们进行对战。我们只需要使得得到随便的坐标值，并且在该坐标中进行下棋操作，就可以实现“电脑”的下棋，因此，实现这个随机下棋的功能，最主要是产生随机的坐标。我们可以使用 `Math.random` 方法来产生 0.0 到 1.0 之间的 `double` 数组，再使用该值来乘以棋盘的大小，即可产生随机的坐标。

我们使用这个方式来产生随机坐标，因为是随机生成的位置，所以这个计算机是比较“笨”的。如果想让计算机变“聪明”起来，可以加上人工智能“电脑”，该人工智能“电脑”需要分析玩家的所有下棋的位置，并对这些位置的坐标进行相应的分析。以下是随机生成坐标的代码实现。

代码清单：`code\gobang\src\org\crazyit\gobang\GobangGame.java`

```

//随机生成 x 坐标，即二维数组具体一维的值
int posX = (int)(Math.random() * ( Chessboard.BOARD_SIZE - 1 ));
//随机生成 y 坐标，即二维数组具体二维的值
int posY = (int)(Math.random() * ( Chessboard.BOARD_SIZE - 1 ));
String[][] board = chessboard.getBoard();
//当棋盘中的位置不是“十”的时候（已经有棋子），则再次生成新的坐标
while( board[posX][posY] != "十" ) {
    posX = (int)(Math.random() * ( Chessboard.BOARD_SIZE - 1 ));
    posY = (int)(Math.random() * ( Chessboard.BOARD_SIZE - 1 ));
}

```

这里需要注意的是，由于我们使用了 `while` 循环，其中循环条件是判断棋盘数组中是否已经存在棋子，如果已经存在棋子，则需要重新随机生成坐标。那么就会产生这样一种情况，如果整个棋盘中都存在棋子的话，这个 `while` 将永远不会跳出，即死循环，所以我们需要判断棋盘是否所有的位置都有棋子，如果棋盘已经都存在棋子并且没有输赢的话，就可以提示玩家和棋了，重新开始游戏。

上面代码中，随机生成 x 与 y 坐标的过程是先用 `Math.random()` 方法获取一个在 `BOARD_SIZE`(棋盘大小)范围内的 x 与 y 正数坐标，如果这个坐标中已经有棋子，则继续使用 `Math.random()` 方法获取坐标，直到这个坐标中没有棋子。



### 1.6.5 是否重新游戏

实现是否重新开始游戏的功能，这在这方法中，程序的流程是：如果玩家或者电脑赢了，则在控制台输出询问玩家是否重新开始游戏的信息，如果玩家输入“y”字符串，则重新开始游戏，否则直接退出整个程序，实现代码如下。

代码清单：code\gobang\src\org\crazyit\gobang\GobangGame.java

```
/**
 * 是否重新开始下棋。
 * @param chessman "●"为用户，"○"为计算机。
 * @return 开始返回 true，反则返回 false。
 */
public boolean isReplay( String chessman )
    throws Exception {
    chessboard.printBoard();
    String message = chessman.equals(Cheesman.BLACK.getChessman())
        ? "恭喜您，您赢了，": "很遗憾，您输了，";
    System.out.println( message + "再下一局? (y/n)" );
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    if( br.readLine().equals("y") ) {
        //开始新一局
        return true;
    }
    return false;
}
```

### 1.6.6 游戏过程实现

以下是游戏的流程说明，具体也可以看图 1.2:

- ❑ （1）首先调用 Chessboard 类型的 chessboard 属性中的 initBoard()与 printBoard()方法去初始化与打印棋盘。
- ❑ （2）从控制台获取用户的输入。
- ❑ （3）再调用本类的 isValid()方法去验证玩家输入的合法性，如果输入不合法，返回第 2 步继续，否则到第 4 步。
- ❑ （4）把玩家下的棋子位置赋值为“●”。
- ❑ （5）调用 isWon( int posX , int posY , String chessman )判断玩家是否赢了。如果玩家赢了，则调用 isReply()方法输出的信息询问玩家是否重新游戏，如果玩家输入 y，则返回第 1 步重新开始。
- ❑ （6）调用 computerDo()方法随机生成计算机的 x,y 坐标，并把 board[x][y] 赋值为“○”。如果计算机赢了，则调用 isReply()方法输出的信息询问玩家是否重新游戏，如果玩家输入 y，则返回第 1 步重新开始，否则返回第 2 步轮到用户输入。

以下的代码实现以上的流程。

代码清单：code\gobang\src\org\crazyit\gobang\GobangGame.java

```
//true 为游戏结束
boolean isOver = false;
chessboard.initBoard();
chessboard.printBoard();
```

```

//获取键盘的输入
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String inputStr = null;
//br.readLine:每当键盘输入一行内容按回车键, 则输入的内容被 br 读取到
while( (inputStr = br.readLine()) != null ) {
    isOver = false;
    if( !isValid( inputStr ) ) {
        //如果不合法, 要求重新输入, 再继续
        continue;
    }
    //把对应的数组元素赋为"●"
    String chessman = Chessman.BLACK.getChessman();
    chessboard.setBoard( posX , posY , chessman );
    //判断用户是否赢了
    if( isWon( posX , posY , chessman ) ) {
        isOver = true;
    } else {
        //计算机随机选择位置坐标
        int[] computerPosArr = computerDo();
        chessman = Chessman.WHITE.getChessman();
        chessboard.setBoard( computerPosArr[0] , computerPosArr[1] , chessman );
        //判断计算机是否赢了
        if( isWon( computerPosArr[0] , computerPosArr[1] , chessman ) ) {
            isOver = true;
        }
    }
}
//如果产生胜者, 询问用户是否继续游戏
if( isOver ) {
    //如果继续, 重新初始化棋盘, 继续游戏
    if( isReplay( chessman ) ) {
        chessboard.initBoard();
        chessboard.printBoard();
        continue;
    }
    //如果不继续, 退出程序
    break;
}
chessboard.printBoard();
System.out.println("请输入您下棋的坐标, 应以 x,y 的格式输入: ");
}

```

以上的代码中, 我们使用了一个 `isOver` 来标识游戏是否胜利, 当游戏胜利时, 就询问玩家是否继续, 我们可以看到, 以上的代码中我们写了多个 `if` 语句, 其实我们可以使用一些 **Java** 的基础知识来解决这些 `if` 问题 (可以使用设计模式中的策略模式), 当然大家也可能觉得这些 `if` 没有什么关系, 但是, 由于这些 `if` 的存在, 会使得我们程序的可读性变差, 在“仿 QQ 游戏大厅”章节, 同样出现了读取字符串关作相应判断的情况, 我们在该章节使用了其他方式去解决这些 `if` 语句, 详细可以看“仿 QQ 游戏大厅”一章。本章的目的主要介绍一个简单五子棋的实现。

## 1.7 本章小结

本章主要是介绍开发控制台五子棋的整个过程，体现流程设计与类设计的基本方法，示范了数组的使用，获取用户键盘的输入。使用了分支结构与循环结构的流程控制，还介绍与使用了枚举类。向读者灌输了面向对象的一些基本知识，通过这些基础的知识设计与开发出有趣味性的小游戏，让读者加深对这些基础知识的理解。

## 第 2 章 仿Windows计算器

### 2.1 仿Windows计算器概述

Windows 计算器，是 Windows 操作系统自带计算器，可以帮助用户完成数据的运算，它可分为“标准型”和“科学型”，本章的仿 Windows 计算器是标准型的 Java 实现，标准型 Windows 计算器实现的主要功能有：四则运算；求倒数；求开方；存储计算结果；读取计算结果；累积计算结果。

我们在第一章中，我们实现了一个在控制台进行的五子棋游戏，我们从本章开始将在 Swing 界面中实现本书的项目。在本章中，我们将使用到 JFrame 和 JPanel 两个 Swing 容器，使用到 JTextField 和 JButton 两个 Swing 容器，使用 BorderLayout 和 GridLayout 做两个布局器，以及使用到事件、事件监听器和事件适配器等。

实现一个计算器，界面中需要提供各种输入的按钮，再以这些按钮组成计算器的键盘，用户点击键盘输入值后，就可以将其所输入的值显示到一个文本框中，运算后，再将结果显示到文本框中。计算器的最终效果如图 2.1 所示。



图 2.1 用 Swing 制作的计算器

从图 2.1 中可以看到，我们开发界面的时候，需要提供一个文本框在窗口的最上部，文本框下面再提供各个计算器的按钮。

#### 2.1.1 数学与其它符号介绍

在此计算器中，主要使用的数学运算有加、减、乘、除四则运算，或者对一个正数进行开方，或者对一个非 0 的数学求倒数，使用到的数学符号有：

- ❑ 加、减、乘、除，对应使用的符号是“+”、“-”、“\*”、“/”。
- ❑ 开方与倒数，对应使用的符号是“sqrt”和“1/x”。
- ❑ 求结果使用的数学符号是“=”。

- ❑ “%”号，如果使用此符号，第二个操作数就等于两数相乘再除以 100。

除了用于数学运算的符号，Windows 计算器还提供对计算结果做存储、读取、累加、清除等操作，亦有对数字显示框中的数字做退格操作，还可以清除上次计算结果或者全部结果：

- ❑ 使用符号“MC”、“MR”、“MS”、“M+”代表清除存储结果、读取存储结果、保存存储结果和累加存储结果。
- ❑ 使用“Backspace”符号代表退格。
- ❑ 使用“CE”和“C”代表清除上次计算结果和清除所有计算结果。

四则运算在程序中可以直接使用 Java 运算符实现，实现开方可以调用 Math 类的 sqrt 方法，倒数可以使用 1 来除以原始的数字。当用户需点击“=”的时候，计算器就需要将最终的计算结果显示到文本框中。其他的计算器功能都可以通过计算器内部的程序实现，例如使用某个字符串或者数字来保存相应的结果，如果需要计取、存储、累加或者清除结果，可以通过改变或者读取我们所保存的值来实现。

### 2.1.2 界面说明

界面中使用的 Swing 组件相对简单，整个大窗口可以看作一个 JFrame 对象，在 JFrame 对象中，存放一个 JPanel 对象，我们需要为这个 JPanel 对象进行布局，将文本框（JTextField 对象）与各个计算器按钮（JButton 对象）添加到这个 JPanel 中。在添加计算器按钮的时候，我们可以使用 GridLayout 布局处理器来进行网格状布局，由于各个计算器按钮都是以网格状分布在界面中的，因此使用 GridLayout 非常适合。本章计算器的界面布局并不复杂，因此在这里不再详细描述。

## 2.2 流程描述

用户打开计算器后，在没有关闭计算器之前，可以通过鼠标点击“1”到“9”数字键和点击“+”、“-”、“\*”、“/”键去输入要运算结果的算术式，再通过点击“=”、“sqrt”、“1/x”等键去直接获取计算结果，除外，还可以点击“MC”、“MR”、“MS”、“M+”键去清除、读取、保存、累加计算显示框中显示的数字，还有清除上次结果、清除所有结果、退格等操作。从图 2.2 中可以看出，计算器打开之后，就开始监听用户的鼠标动作，如果输入是关于计算结果或者“MC”、“MR”、“MS”、“M+”、“Backspace”、“CE”、“C”等操作指令，而且没有关闭计算器，就返回计算结果并显示，如果不是，则不计算结果。接下来再继续等待用户的输入。

本章的计算器并没有复杂的流程，只需要简单的操作，返回计算结果等。在实现计算器的过程中，我们需要注意的是，例如已经点击了某个数字，再点击运算符，那么程序需要记录之前选点击的数字，当用户再次点击运算符（非“=”）时，系统就需要将结果显示到文本框中。因此在开发计算器的时候，我们需要注意用户点击的具体顺序。

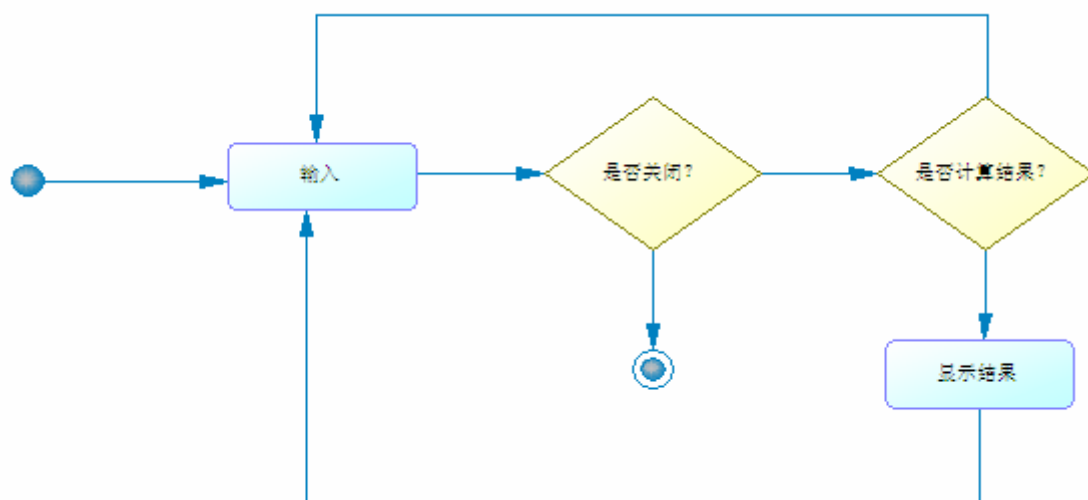


图 2.2 计算流程

## 2.3 建立计算器对象

实现一个计算器，我们需要建立一系列的对象来实现，例如，计算界面我们要建立一个界面类，还需要建立一个专门负责处理加、减、乘、除的基本计算类，还需要一个负责处理计算功能的业务类。本小节中只讲解创建这三个基本的类，如果在开发的过程发现可以将一些行为或者属性放置到一个新的对象中，那么可以再建立这些对象来完成需要实现的功能或者操作。

本章主要设计四个类来完成计算器的功能，界面类（**CalFrame**）——主要用来显示计算器界面，功能类（**CalService**）——主要用于完成计算器中的逻辑功能，计算工具类（**MyMath**）——此类是工具类，用于处理大型数字的加减乘除，计算器类（**Cal**）——用于打开计算器，计算器中各个类的关系如图 2.3 所示，从图中可以看出，我们的界面类继承了 `java.swing.JFrame` 类，计算器类使用了界面类，界面类使用了功能类，功能类使用了 **MyMath** 工具类，下面章节将对这些计算器的相关类作详细介绍。

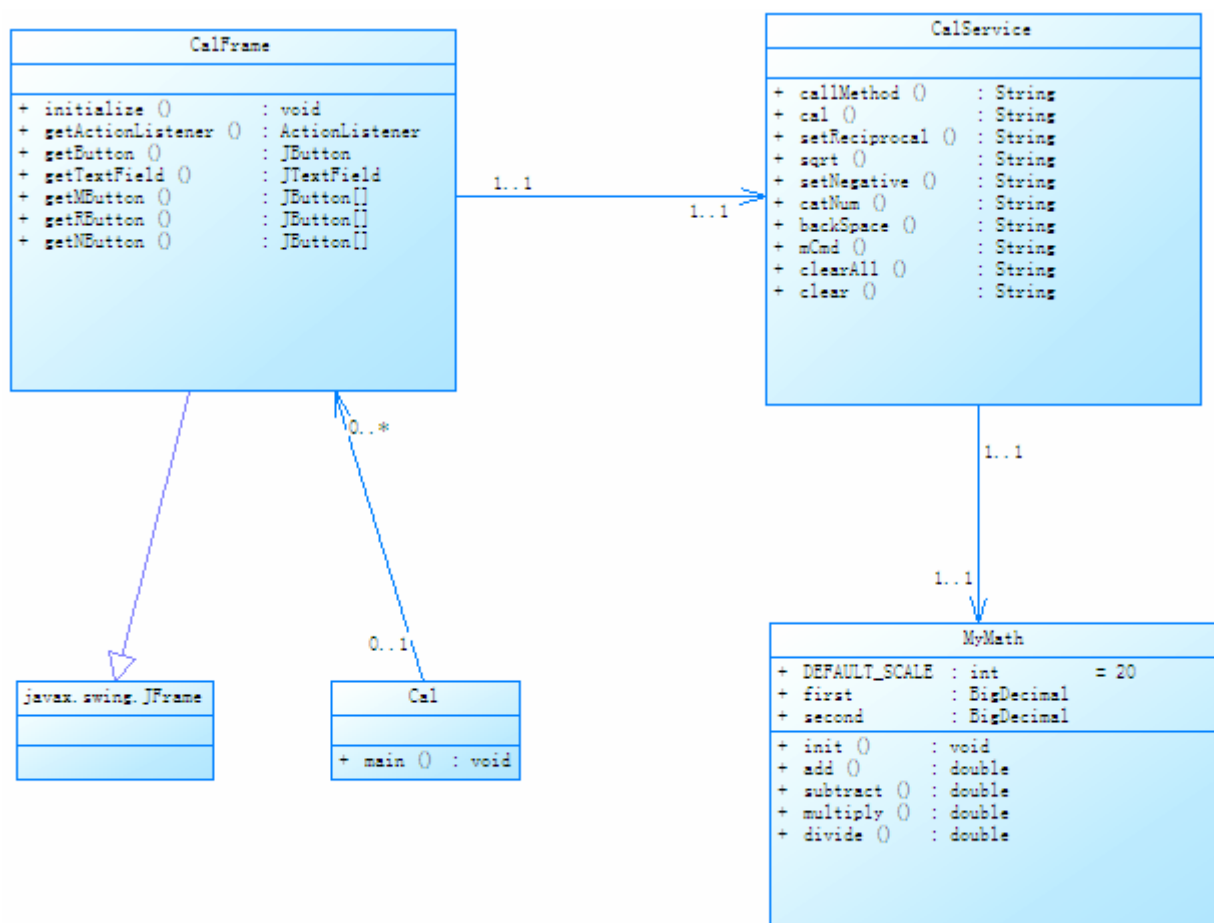


图 2.3 计算器类图

### 2.3.1 MyMath工具类

使用 `float`, `double` 两种浮点基本类型来进行计算，容易损失精度，所以，我们使用一个自己定义了加，减，乘，除方法的类，此类使用 `BigDecimal` 来封装基本类型，在不损失精度的同时，也可以进行超大数字的四则运算。为了方便调用，此类的方法全部都是静态方法，可以直接用“类名.方法名”调用，这个类包含以下方法：

- ❑ `static double add( double num1, double num2 )`，加法，使用来计算结果的数字是封装后的 `num1` 和 `num2`，并返回 `double` 类型。
- ❑ `static double subtract ( double num1, double num2 )`，减法，使用来计算结果的数字是封装后的 `num1` 和 `num2`，并返回 `double` 类型。
- ❑ `static double multiply ( double num1, double num2 )`，乘法，使用来计算结果的数字是封装后的 `num1` 和 `num2`，并返回 `double` 类型。
- ❑ `static double divide ( double num1, double num2 )`，除法，使用来计算结果的数字是封装后的 `num1` 和 `num2`，并返回 `double` 类型。

**MyMath** 类提供了基础的四则运算方法，由于该类中所有的方法都是静态的，因此外界可以直接调用。在实现 **MyMath** 的过程中需要注意的是，这几个四则运算方法，参数都是 `double` 类型的，要进行运算的话，需要将 `double` 类型转换成一个 `BigDecimal` 对象，我们可以使用以下代码来创建一个 `BigDecimal` 对象：

```
new BigDecimal(String.valueOf(number));
```

### 2.3.2 CalService类

CalService 类主要是用来处理计算器的业务逻辑，用户在操作计算器时，此类将计算结果，并且返回，并且，会记录计算器的状态（用户的上一步操作）。包含以下方法：

- ❑ String callMethod( String cmd , String text )，调用方法并返回计算结果。
- ❑ String cal( String text , boolean isPercent )，用来计算加、减、乘、除法，并返回封装成 String 内型的结果。参数 text 是显示框中的数字内容，boolean 类型的参数 isPercent 代表是否有“%”运算，如果有，便加上去。
- ❑ String setReciprocal( String text )，用来计算倒数，并返回封装成 String 内型的结果。
- ❑ String sqrt( String text )，用来计算开方，并返回封装成 String 内型的结果。
- ❑ String setOp( String cmd , String text )，设置操作符号。
- ❑ String setNegative( String text )，设置正负数，当 text 是正数时，返回负数的数字字符串，反之，则返回正数的数字字符串。
- ❑ String catNum( String cmd, String text )，连接输入的数字，每次点击数字，就把把新加的数字追加到后面，并封装成字符串返回。
- ❑ String backSpace( String text )，删除最后一个字符，并返回结果。
- ❑ String mCmd( String cmd, String text )，用来实现“M+”、“MC”、“MR”、“MS”与存储有关的功能。
- ❑ String clearAll()，清除所有计算结果。
- ❑ String clear( String text)，清除上次计算结果。

CalService 类中的各个方法都是用于处理计算的逻辑，其中 callMethod 方法可以看作中一个中转的方法，根据参数中的 cmd 值进行分发处理，例如调用该方法时将“CE”字符串作为 cmd，那么该方法就根据这个字符串再调用需要执行“CE”的方法。如果需要做更好的程序解耦，我们可以将这些做成一个状态模式，将各个计算的方法都抽象成一个计算接口，该接口提供一个计算的方法，然后按照具体的情况，为该接口提供不同的实现，例如计算开方、计算倒数等实现，然后向 callMethod 传入不同的实现类，直接调用接口方法。

### 2.3.3 CalFrame类

CalFrame 类继承 javax.swing.JFrame 类，主要是用于计算器界面的实现，此类中，排版了计算器中各个组件的位置，为组件增加事件监听器，用来监听用户的操作，并做调用相应的方法，主要包含以下方法：

- ❑ void initialize()，初始化计算器界面。
- ❑ ActionListener getActionListener()，如果动作监听器为空，则创建一个，并返回，如果不为空，直接返回。
- ❑ JTextField getTextField()，这个方法初始化输入框。
- ❑ JButton[] getMButton()，此方法获得计算器的存储操作键。
- ❑ JButton[] getRButton()，此方法获得计算器的结果操作键。
- ❑ JButton[] getNButton()，此方法获得计算器的其它操作键。

由于 CalFrame 是界面类，因此所需要进行的业务处理并不多，更多的是监听用户的操作，并进行分发处理。这就有点像 web 应用中的 MVC 模式中的 V（视图），并不处理任务的业务逻辑，主要职责



是显示相应的数据。在本章中，**CalFrame** 包括了一些监听器，监听界面事件并调用相关的业务方法，在实际开发中，我们可以将这些监听器作为 MVC 模式中的 C（控制器）提取到另外的类中。

## 2.4 MyMath工具类实现

**MyMath** 是一个工具类，主要用于处理加、减、乘、除四则运算，我们已经确定了实现这四个方法的时候，都使用 **BigDecimal** 对象进行计算。由于我们定义 **MyMath** 方法的时候，所有方法的参数都是 **double** 类型的，因此我们可以提供一个工具方法来将 **double** 转换成 **BigDecimal** 类型。

以下代码根据一个 **double** 类型转换成一个 **BigDecimal**。

代码清单：code\cal\src\org\crazyit\cal\MyMath.java

```
/**
 * 为一个 double 类型的数字创建 BigDecimal 对象
 * @param number
 * @return
 */
private static BigDecimal getBigDecimal(double number) {
    return new BigDecimal(number);
}
```

提供了这个工具方法后，我们可以在其他的计算方法中使用这个工具方法，选择将 **double** 的参数转换成 **BigDecimal** 对象，然后再进行具体的运算。

### 2.4.1 实现四则运算

编写了 **double** 转换的工具方法后，实现加、减、乘、除比较简单，由于 **BigDecimal** 已经为我们实现了，因此可以直接调用该类的相应方法即可实现，以下代码分别实现四则运算。

代码清单：code\cal\src\org\crazyit\cal\MyMath.java

加法：

```
public static double add(double num1, double num2) {
    //调用工具方法将 double 转换成 BigDecimal
    BigDecimal first = getBigDecimal(num1);
    BigDecimal second = getBigDecimal(num2);
    return first.add(second).doubleValue();
}
```

减法：

```
public static double subtract(double num1, double num2) {
    BigDecimal first = getBigDecimal(num1);
    BigDecimal second = getBigDecimal(num2);
    return first.subtract(second).doubleValue();
}
```

乘法：

```
public static double multiply(double num1, double num2) {
    BigDecimal first = getBigDecimal(num1);
    BigDecimal second = getBigDecimal(num2);
    return first.multiply(second).doubleValue();
}
```

除法：

```
public static double divide(double num1, double num2) {
```

```
BigDecimal first = getBigDecimal(num1);
BigDecimal second = getBigDecimal(num2);
return first.divide(second, DEFAULT_SCALE, BigDecimal.ROUND_HALF_UP)
    .doubleValue();
}
```

四个方法都是调用了 **BigDecimal** 的方法来实现，Java 的 **BigDecimal** 类为我们提供了许多强大的计算方法，可以让我们很方便的进行数学运算，除本章介绍的方法外，读者可以查阅 Java 的 API 来学习该类的详细使用。

## 2.5 计算器主界面

这里实现计算器的界面，是用 java 的 **Swing** 实现的，主要用到的类有 **javax.swing.JFrame**（窗口），**javax.swing.JButton**（按钮），**javax.swing.JTextField**（输入框），并使用 **java.awt.BorderLayout** 和 **java.awt.GridLayout** 进行布局。在这里，我们使用“自下而上”的方法去观察此类，先看总体的排版实现，再看各个小组件的实现。为了方便布局，我们按相近的外观把计算器分为四个部分，见图 2.4：



图 2.4 布局

### 2.5.1 初始化界面（initialize()方法）

此类就是一个 **JFrame**（继承了 **javax.swing.JFrame**），用来做其它窗口或者组件的父容器，初始化计算器窗口的大概流程：

- ❑ 设置父窗口 **JFrame** 标题、布局管理器、是否可以改变等属性。
- ❑ 增加输入与计算结果显示框。对应图 2.4 中的左上角那部分。
- ❑ 增加左边存储操作键。
- ❑ 增加结果操作键。
- ❑ 增加数字与其它运算符。

由于按外观相近的方式把组件分成了四部分，就方便程序中对相同属性的组件统一地创建与设置属性，对于界面的布局也更加地直观与方便，观察此图，我们可以使用 **BorderLayout** 做总体布局，如图 2.5 所示。

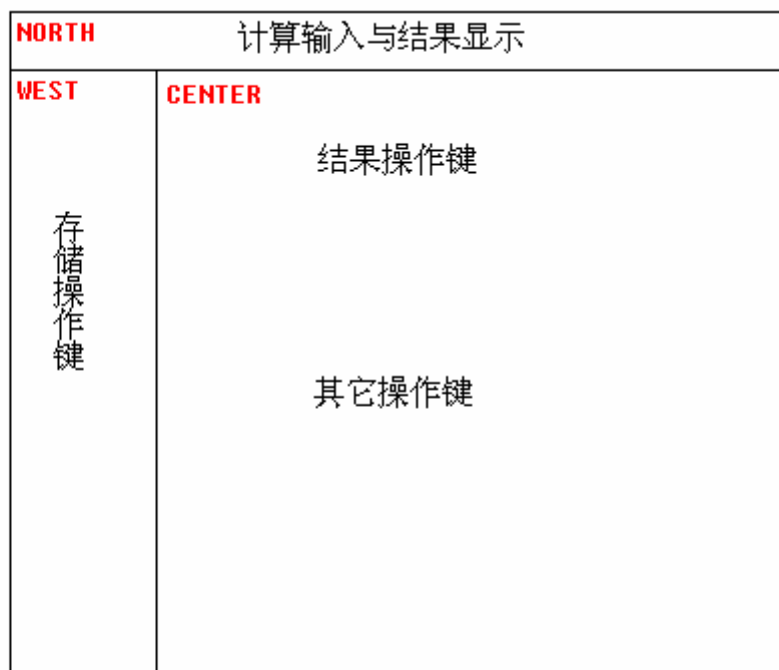


图 2.5 布局管理器

以下代码设置父窗口 **JFrame** 标题和设置是否可以改变大小的属性。

```
//设置窗口的标题
```

```
this.setTitle("计算器");
```

```
//设置为不可改变大小
```

```
this.setResizable( false );
```

增加输入与结果显示的 **TextField** 输入框，这里调用本类的 **getTextField()** 方法获取，并把它加入 **panel** 中的 **NORTH** 位置中：

```
//增加计算输入框
```

```
JPanel panel = new JPanel();
```

```
panel.setLayout( new BorderLayout(10,1) );
```

```
panel.add( getTextField(), BorderLayout.NORTH );
```

```
panel.setPreferredSize( new Dimension( PRE_WIDTH, PRE_HEIGHT ) );
```

增加左边存储操作键，本类需要通过 **getMButton()** 方法获取一个保存  **JButton**  对象的数组，**getMButton** 方法我们将在 2.5.2 中实现。获取数组后，遍历数组，并把数组中的元素加到一个新建的 **JPanel** 中，最后再把这个 **JPanel** 加到 **JFrame** 的相应位置：

```
//增加左边存储操作键
```

```
JButton[] mButton = getMButton();
```

```
//新建一个 panel，用于放置按钮
```

```
JPanel panel1 = new JPanel();
```

```
//设置布局管理器
```

```
panel1.setLayout( new GridLayout( 5, 1, 0, 5 ) );
```

```
//迭代增加按钮
```

```
for( JButton b : mButton ) panel1.add(b);
```

增加结果操作键，这些结果操作键包括：**Back**，**CE**，**C**。通过本类的 **getRButton()** 方法获取一个保存  **JButton**  对象的数组，获取数组后，遍历数组，并把数组中的元素加到一个新建的 **JPanel** 中，最后再把这个 **JPanel** 加到 **JFrame** 相应的位置，具体实现的代码如下：

```
//增加结果操作键
```

```

JButton[] rButton = getRButton();
JPanel panel2 = new JPanel();
panel2.setLayout( new BorderLayout(1, 5) );
//新建一个 panel, 用于放置按钮
JPanel panel21 = new JPanel();
//设置布局管理器
panel21.setLayout( new GridLayout( 1, 3, 3, 3 ) );
//迭代增加按钮
for( JButton b : rButton ) panel21.add(b);

```

接下来将其他的按钮加入到界面的 `JPanel` 对象中, 这些操作键主要包括数字键和其他的一些运算键, 我们同样的通过一个 `getNButton` 方法来返回这些操作键对应的 `JButton` 对象, 最后将这些 `JButton` 对象加入到相应的 `JPanel` 中, 加入到 `JPanel` 并设置相应布局的代码如下:

```

//增加数字与其它运算符
JButton[] nButton = getNButton();
//新建一个 panel, 用于放置按钮
JPanel panel22 = new JPanel();
//设置布局管理器
panel22.setLayout( new GridLayout( 4, 5, 3, 5 ) );
//迭代增加按钮
for( JButton b : nButton ) panel22.add(b);
//把新增加的面板加到 frame
...
this.add(panel);

```

在本小节中, 我们通过 `getMButton`、`getRButton` 和 `getNButton` 方法来返回不同的 `JButton` 数组, 然后再对这些数组进行遍历, 将每一个 `JButton` 加入到界面中。这一个返回 `JButton` 数组的方法并没有实现, 下面将介绍如何实现这三个方法。

以上所有的代码均在 `code\cal\src\org\crazyit\cal\CalFrame.java` 中。

## 2.5.2 创建运算键

运算键主要包括数字键与基本运算键, 数字键从 0 到 9, 基本运算键包括开方、正负、小数点等键, 主要实现计算器界面的 `getNButton` 方法即可。以下是该方法的实现。

代码清单: `code\cal\src\org\crazyit\cal\CalFrame.java`

```

private JButton[] getNButton() {
    // 这个数组保存需要设置为红色的操作符
    String[] redButton = { "/", "**", "-", "+", "=" };
    JButton[] result = new JButton[nOp.length];
    for (int i = 0; i < this.nOp.length; i++) {
        // 新建按钮
        JButton b = new JButton(this.nOp[i]);
        // 为按钮增加事件
        b.addActionListener(getActionListener());
        // 对 redButton 排序, 才可以使用 binarySearch 方法
        Arrays.sort(redButton);
        // 如果操作符在 redButton 出现
        if (Arrays.binarySearch(redButton, nOp[i]) >= 0) {
            b.setForeground(Color.red);
        } else {
            b.setForeground(Color.blue);
        }
    }
}

```

```

    }
    result[i] = b;
}
return result;
}

```

以上代码需要注意的是，我们需要提供一个红色按键的字符串数组，在遍历所有的需要创建的按键数组时，就需要作判断，如果按键数组里面存在红色按键数组的某个元素，就需要调用 `JButton` 的 `setForeground` 方法来设置该按钮的字体颜色。在代码中我们不能看到该方法帮我们创建了哪些按键，代码中使用了一个 `nOp` 的字符串数组来保存需要创建的按键，该数组包含的内容如下：

```

private String[] nOp = { "7", "8", "9", "/", "sqrt", "4", "5", "6", "**",
    "%", "1", "2", "3", "-", "1/x", "0", "+/-", ".", "+", "=" };

```

### 2.5.3 创建操作按键

操作按键的创建与运算键的创建基本一致，只是所有的按键的字体都必须是红色的，创建操作按钮，我们需要实现 `getMButton` 和 `getRButton` 方法，以下是这两个方法的具体实现。

代码清单：code\cal\src\org\crazyit\cal\CalFrame.java

```

private JButton[] getMButton() {
    JButton[] result = new JButton[mOp.length + 1];
    result[0] = getButton();
    for (int i = 0; i < this.mOp.length; i++) {
        // 新建按钮
        JButton b = new JButton(this.mOp[i]);
        // 为按钮增加事件
        b.addActionListener(getActionListener());
        // 设置按钮颜色
        b.setForeground(Color.red);
        result[i + 1] = b;
    }
    return result;
}

private JButton[] getRButton() {
    JButton[] result = new JButton[rOp.length];
    for (int i = 0; i < this.rOp.length; i++) {
        // 新建按钮
        JButton b = new JButton(this.rOp[i]);
        // 为按钮增加事件
        b.addActionListener(getActionListener());
        // 设置按钮颜色
        b.setForeground(Color.red);
        result[i] = b;
    }
    return result;
}

```

`getMButton` 创建的是界面左侧的操作键，`getRButton` 创建的是运作键上面的操作键，`getMButton` 和 `getRButton` 创建的操作键如下：

```

//getMButton
private String[] mOp = { "MC", "MR", "MS", "M+" };
//getRButton

```

```
private String[] rOp = { "Back", "CE", "C" };
```

创建完界面元素后，我们可以运行计算器，具体的效果如图 2.4 所示。

### 2.5.4 增加事件监听器

在上一节中，我们注意到程序为 `JButton` 类型的组件增加了事件监听器，这个事件监听器是用来响应用户的鼠标操作。我们使用 `java.awt.event.ActionListener` 接口来创建一个事件监听器，主要是实现接口中的 `actionPerformed( ActionEvent e )` 方法，当监听器监听到用户的操作时，会自动调用此方法，并在此方法中处理业务逻辑，再把数据返回显示给用户。见以下代码。

代码清单：code\cal\src\org\crazyit\cal\CalFrame.java

```
actionListener = new ActionListener(){
    public void actionPerformed( ActionEvent e ) {
        String cmd = e.getActionCommand();
        String result = null;
        try {
            //计算操作结果
            result = service.callMethod( cmd, textField.getText() );
        } catch( Exception e1 ) {
            System.out.println( e1.getMessage() );
        }
        //处理 button 的标记
        if( cmd.indexOf("MC") == 0 ) {
            button.setText("");
        } else if( cmd.indexOf("M") == 0 && service.getStore() > 0 ) {
            button.setText("M");
        }
        //设置计算结果
        if( result != null ) {
            textField.setText( result );
        }
    }
};
```

从上面代码中可以看到，这里是通过实现 `java.awt.event.ActionListener` 接口中的 `actionPerformed( ActionEvent e )` 方法去创建一个 `java.awt.event.ActionListener` 类型的内部类，并在 `actionPerformed` 方法中处理业务逻辑。

首先，调用 `CalService` 实例中的 `callMethod` 方法去处理计算，并把结果返回。

```
result = service.callMethod( cmd, textField.getText() );
```

再设置标志存储结果类型的存储标记，如果是点击“MC”按钮，就把标记设置为空，如果是点击“MS”，“MR”，“M+”，并且存储结果大于 0，就把标记设置为“M”，这里弄不明白的读者，可以先试着使用一下 windows 计算器的这几个按钮，再看这里就很容易理解了。

```
if( cmd.indexOf("MC") == 0 ) {
    button.setText("");
} else if( cmd.indexOf("M") == 0 && service.getStore() > 0 ) {
    button.setText("M");
}
```

最后把计算结果设置到结果文本显示框中，显示给使用者。

```
if( result != null ) {
    textField.setText( result );
}
```

在监听器中，我们调用了 `CalServer` 的 `callMethod` 方法来取得操作的结果，换言之，界面中的每次点击都会执行该方法，`callMethod` 我们并没有提供任何实现，在下一小节，我们将实现该方法。

## 2.6 计算业务处理

在 2.3 章节中，我们建立了一个类名为 `CalService` 的类来处理计算器的计算业务，该类处理了整个应用中的大部分业务，其中包括数字运算，存储运算，操作结果等业务。有四个重要的属性：`firstNum` 代表第一个操作数，`secondNum` 代表第二个操作数，`lastOp` 代表上次用户所做的操作，`isSecondNum` 代表是否第二个操作数。

### 2.6.1 计算四则运算结果

在使用计算器计算加、减、乘、除法的过程中，正常的情况应该是用户先输入第一个操作数，再点击加、减、乘、除计算符号，再输入第二个操作数，最后点“=”号计算出结果，所以这时用 `firstNum` 去保存用户输入的第一个操作数，`secondNum` 去保存第二个操作数，`lastOp` 去保存计算符号或者其它操作，`isSecondNum` 用来判断用户是在输入第几个操作数。

在用户输入数字的时候（包括“0123456789.”），首先判断是第一个操作数还是第二个，如果是第一个，就把用户新输入的数字追加到原来数字的后面，并做为结果返回；如果是第二个，直接返回结果，并把 `isSecondNum` 标志为 `false`，用户继续输入数字的时候，就把数字追加到原来数字的后面做为结果返回，见以下代码。

代码清单：code\cal\src\org\crazyit\cal\CalService.java

```
public String catNum( String cmd, String text ) {
    String result = cmd;
    //如果目前的 text 不等于 0
    if( !text.equals("0") ) {
        if( isSecondNum ) {
            //将 isSecondNum 标志为 false
            isSecondNum = false;
        } else {
            //刚返回结果为目前的 text 加上新点击的数字
            result = text + cmd;
        }
    }
    //如果有.开头，刚在前面补 0
    if( result.indexOf(".") == 0 ) {
        result = "0" + result;
    }
    return result;
}
```

当用户点击“+、-、\*、/”（四则运算）的时候，就把 `lastOp` 设置为其中一个符号，这个变量用来记录用户正要进行计算的类型，见以下代码。

代码清单：code\cal\src\org\crazyit\cal\CalService.java

```
public String setOp( String cmd , String text ) {
    //将此操作符号设置为上次的操作
    this.lastOp = cmd;
    //设置第一个操作数的值
    this.firstNum = text;
}
```

```

//将第二个操作数赋值为空
this.secondNum = null;
//将 isSecondNum 标志为 true
this.isSecondNum = true;
//返回空值
return null;
}

```

在上面的代码中, 可以看到, 除了设置 `lastOp` 外, 还把输入的数字设置给 `firstNum`, 把 `secondNum` 设置为空, 并把 `isSecondNum` 设置为 `true`, 代表下次输入数字时, 要清空输入框并重新输入。

最后用户点击 “=” 号时, 就是程序计算出最后结果的时候, 此类的 `String cal( String text , boolean isPercent )` 方法实现了此计算, 注意到这个方法的第二个参数 `isPercent`, 这是计算器的 “%” 号操作, 如果有这种操作, 第二个操作数就等于两数相乘再除以 100, 请看以下代码。

代码清单: `code\cal\src\org\crazyit\cal\CalService.java`

```

public String cal( String text , boolean isPercent )
throws Exception {
//初始化第二个操作数
double secondResult = secondNum == null
    ? Double.valueOf( text ).doubleValue()
    : Double.valueOf( secondNum ).doubleValue();
//如果除数为 0, 不处理
if( secondResult == 0 && this.lastOp.equals("/") ) {
    return "0";
}
//如果有 "%" 操作, 则第二个操作数等于两数相乘再除以 100
if( isPercent ) {
    secondResult = MyMath.multiply( Double.valueOf( firstNum )
        , MyMath.divide( secondResult, 100 ) );
}
//四则运算, 返回结果赋给第一个操作数
if( this.lastOp.equals("+") ) {
    firstNum = String.valueOf(
        MyMath.add( Double.valueOf( firstNum ), secondResult ) );
}
else if( this.lastOp.equals("-") ) {
    firstNum = String.valueOf(
        MyMath.subtract( Double.valueOf( firstNum ), secondResult ) );
} else if( this.lastOp.equals("*") ) {
    firstNum = String.valueOf(
        MyMath.multiply( Double.valueOf( firstNum ), secondResult ) );
}
else if( this.lastOp.equals("/") ) {
    firstNum = String.valueOf(
        MyMath.divide( Double.valueOf( firstNum ), secondResult ) );
}
//给第二个操作数重新赋值
secondNum = secondNum == null ? text : secondNum;
//把 isSecondNum 标志为 true
this.isSecondNum = true;
return firstNum;
}

```



上面计算结果中，经历了几个步骤，首先，确定 `secondNum` 的值，如果 `secondNum` 为空，`secondNum` 就等于最后输入的数字，如果不为空，则等于原来的值，如果有“%”号操作，则 `secondNum` 再等于两数相乘除以 100 的结果；然后根据 `lastOp` 的值( +、-、\*、/ )去调用 `MyMath` 类中的 `add`、`subtract`、`multiply`、`divide` 方法，并把返回的结果保存到 `firstNum`；最后把 `secondNum` 设置为空，把 `firstNum` 当做结果返回。

## 2.6.2 存储操作

定义一个 `double` 类型的属性 `store` 来充当存储器，在用户点击“MC（清除）”、“M+（累加）”、“MR（读取）”、“MS（保存）”操作时，就调用此方法，再根据得到的字符串去进行清除、累加、读取、保存操作，见以下代码。

代码清单：code\cal\src\org\crazyit\cal\CalService.java

```
public String mCmd( String cmd, String text ) {
    if( cmd.equals( "M+" ) ) {
        //如果是"M+"操作,刚把计算结果累积到 store 中
        store = MyMath.add( store, Double.valueOf( text ) );
    } else if( cmd.equals( "MC" ) ) {
        //如果是"MC"操作, 则清除 store
        store = 0;
    } else if( cmd.equals( "MR" ) ) {
        //如果是"MR"操作, 则把 store 的值读出来
        isSecondNum = true;
        return String.valueOf( store );
    } else if( cmd.equals( "MS" ) ) {
        //如果是"MS"操作, 则把计算结果保存到 store
        store = Double.valueOf( text ).doubleValue();
    }
    return null;
}
```

程序中提供了一个 `store` 的属性用来保存计算结果，当用户点击了“M+”时，就将结果加到 `store` 中，点击了“MC”时，就将 `store` 设置为 0，点击了“MR”，则将 `store` 的值读取，点击了“MS”，则将 `store` 设置为当前的结果。

## 2.6.3 实现开方、倒数等

开方与倒数的计算实现都比较简单，开方是直接使用 `Math` 类的 `sqrt` 方法去计算接收到的数值，并且返回结果：

```
public String sqrt(String text) {
    // 将 isSecondNum 标志为 true
    this.isSecondNum = true;
    // 计算结果并返回
    return String.valueOf(Math.sqrt(Double.valueOf(text)));
}
```

倒数是调用 `MyMath` 的 `divide` 方法去计算 1 与接收到的数值相除的值。

代码清单：code\cal\src\org\crazyit\cal\CalService.java

```
public String setReciprocal(String text) {
    // 如果 text 为 0, 则不求倒数
    if (text.equals("0")) {
```

```

        return text;
    } else {
        // 将 isSecondNum 标志为 true
        this.isSecondNum = true;
        // 计算结果并返回
        return String.valueOf(MyMath.divide(1, Double.valueOf(text)));
    }
}

```

### 2.6.4 实现倒退操作

当我们的程序中得到用户在界面输入的相关数字时，如果用户进行了倒退操作，我们可以将用户输入的数字进行截取，如果接收到的字符串是“0”或者为 null，则不作任何操作，直接返回，否则，我们将使用 **String** 的 **substring** 方法进行处理，将输入字符串的最后一位截取。以下方法实现倒退操作。

代码清单：code\cal\src\org\crazyit\cal\CalService.java

```

public String backSpace(String text) {
    return text.equals("0") || text.equals("") ? "0" : text.substring(0,
        text.length() - 1);
}

```

### 2.6.5 清除计算结果

清除所有计算结果，把 **firstNum** 与 **secondNum** 都设置为原始值，并返回 **firstNum**，在 **CalService** 中提供了一个 **clearAll** 方法，用于清除所有的计算结果。

代码清单：code\cal\src\org\crazyit\cal\CalService.java

```

public String clearAll() {
    // 将第一第二操作数恢复为默认值
    this.firstNum = "0";
    this.secondNum = null;
    return this.firstNum;
}

```

### 2.6.6 实现中转方法（callMethod）

在前面的章节中，我们已经实现了各个方法，例如四则运算、开方、倒数、清除计算等，但是在界面的监听器中，只会调用 **CalService** 的 **callMethod** 方法进行运算，因此我们需要对 **callMethod** 进行相关的实现。

代码清单：code\cal\src\org\crazyit\cal\CalService.java

```

public String callMethod(String cmd, String text) throws Exception {
    if (cmd.equals("C")) {
        return clearAll();
    } else if (cmd.equals("CE")) {
        return clear(text);
    } else if (cmd.equals("Back")) {
        return backSpace(text);
    } else if (numString.indexOf(cmd) != -1) {
        return catNum(cmd, text);
    } else if (opString.indexOf(cmd) != -1) {

```

```
        return setOp(cmd, text);
    } else if (cmd.equals("=")) {
        return cal(text, false);
    } else if (cmd.equals("+/-")) {
        return setNegative(text);
    } else if (cmd.equals("1/x")) {
        return setReciprocal(text);
    } else if (cmd.equals("sqrt")) {
        return sqrt(text);
    } else if (cmd.equals("%")) {
        return cal(text, true);
    } else {
        return mCmd(cmd, text);
    }
}
```

CalService 中的 `callMethod` 方法，只是判断输入命令，再决定调用具体的哪个方法处理计算。例如监听器监听到用户点击了倒退了按键，那么 `callMethod` 方法就会根据点击的按键文本来找到 `backSpace` 方法。当然，使用这么多的 `if...else...` 并不是最佳的解决方案，我们可以使用一些的设计模式来解决。有兴趣的读者可以了解相关的设计模式，考虑如何解决这些问题。

## 2.7 本章小结

本章主要是通过一个仿 Windows 计算器的基本实现，向读者讲解 Java swing 编程，示范了 `JFrame`，`JPanel`，`JTextField`，`JButton` 的使用。界面布局方面，使用到了 `awt` 的 `BorderLayout` 与 `GridLayout` 布局管理器去布局。并且向读者介绍了 `ActionLisner` 事件监听器的使用，介绍如何监听用户的动作响应用户，并且向用户返回有用的信息。本章中实现的计算相对较为简单，有兴趣的读者可以在本文的基础上实现更强大的计算器（科学型计算器）。另外需要注意的是，本程序编写的过程中，使用了许多 `if...else...` 语句，对设计模式有一定了解或者希望对此有了解的读者，可以尝试去重构本章的代码，消除这些 `if...else...`。在下面的章节中，我们会在编写的过程中，展示一些设计模式的概念。

## 第3章 图片浏览器

### 3.1 图片浏览器概述

相信使用 Window 操作系统的大多数用户，都使用过 Windows 的图片浏览器，或者是功能更强大与复杂的 ACDSee 图片浏览器（这个还支持编辑图片），图片浏览器最基本的功能是能浏览一个目录中的所有图片，并可以点击浏览上一张图片或者下一张图片，还有对图片放大与缩小，或者翻转图片等操作，在这里，实现了图片的浏览功能，导航功能（下一张、上一张），放大缩小功能。

本章将实现一个最简单的图片浏览器，包括了打开图片、放大与缩小图片、查看上一张和下一张图片等功能，图片浏览器的最终效果如图 3.1 所示。

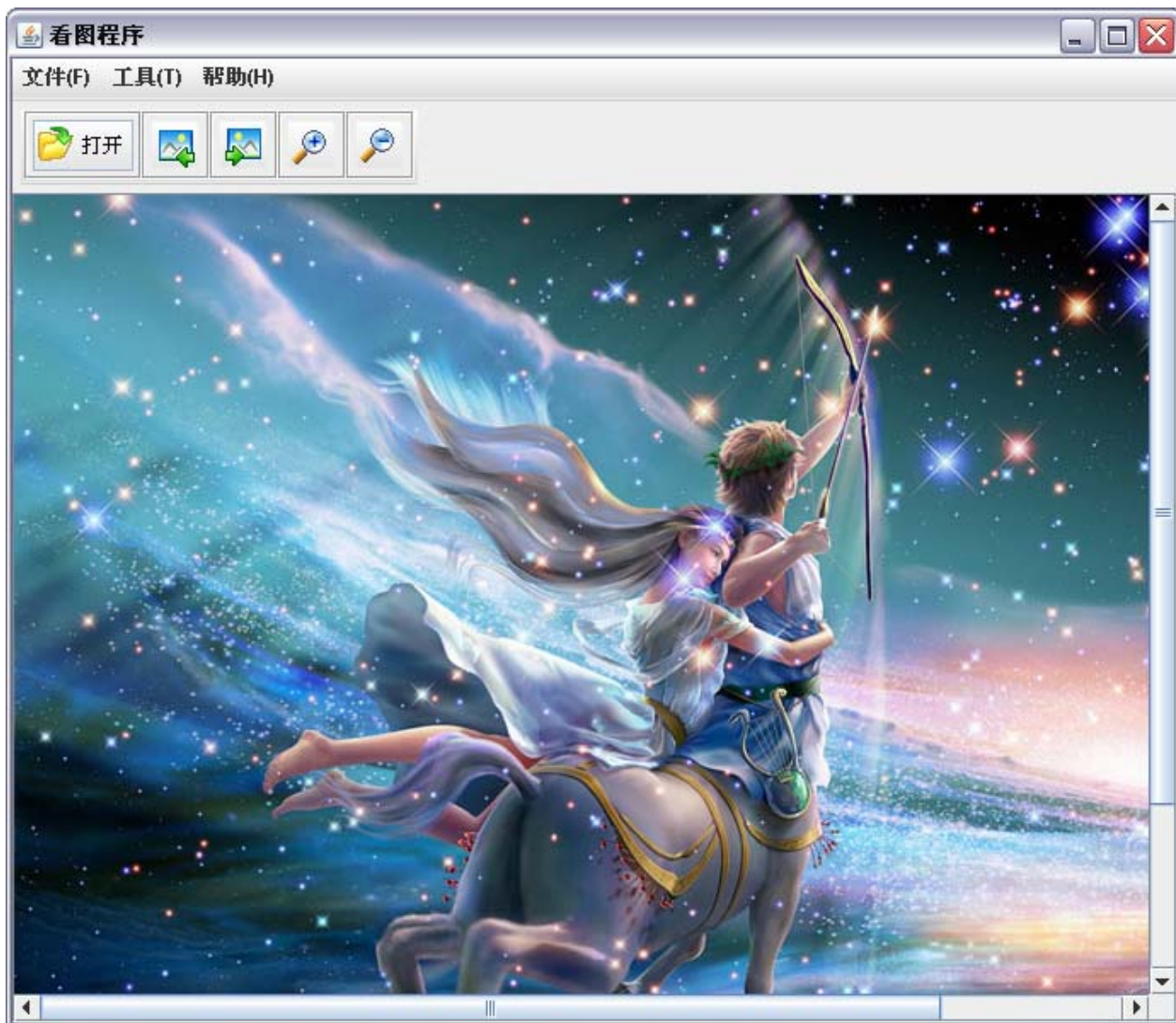


图 3.1 图片浏览器

## 3.2 创建图片浏览器的相关对象

我们首先需要创建图片浏览器的相关对象。我们先创建图片浏览器的界面对象 `ViewerFrame`，然后在类中，我们为菜单、按钮加了事件监听器，所以定义了一个继承 `AbstractAction` 的类 `ViewerAction` 来响应这些动作。在 `Action` 中响应动作，就处理具体逻辑的步骤，我们把所有的逻辑处理放到 `ViewerService` 类中，`ViewerService` 中包括打开图片、上一张、下一张、放大和缩小图片等功能，为了程序更好的解耦合，我们可以把具体的某些业务处理放置到独立的类中进行处理。

除了以上所说的几个类，由于我们这个程序有打开图片的操作，所以需要有一个文件过滤器（只能选择图片类型的文件），所以定义了一个继承 `JFileChooser` 的类 `ViewerFileChooser`，这个类里面定义了自己的文件过滤器。本章中所涉及的对象及它们之间的关系如图 3.2 所示。

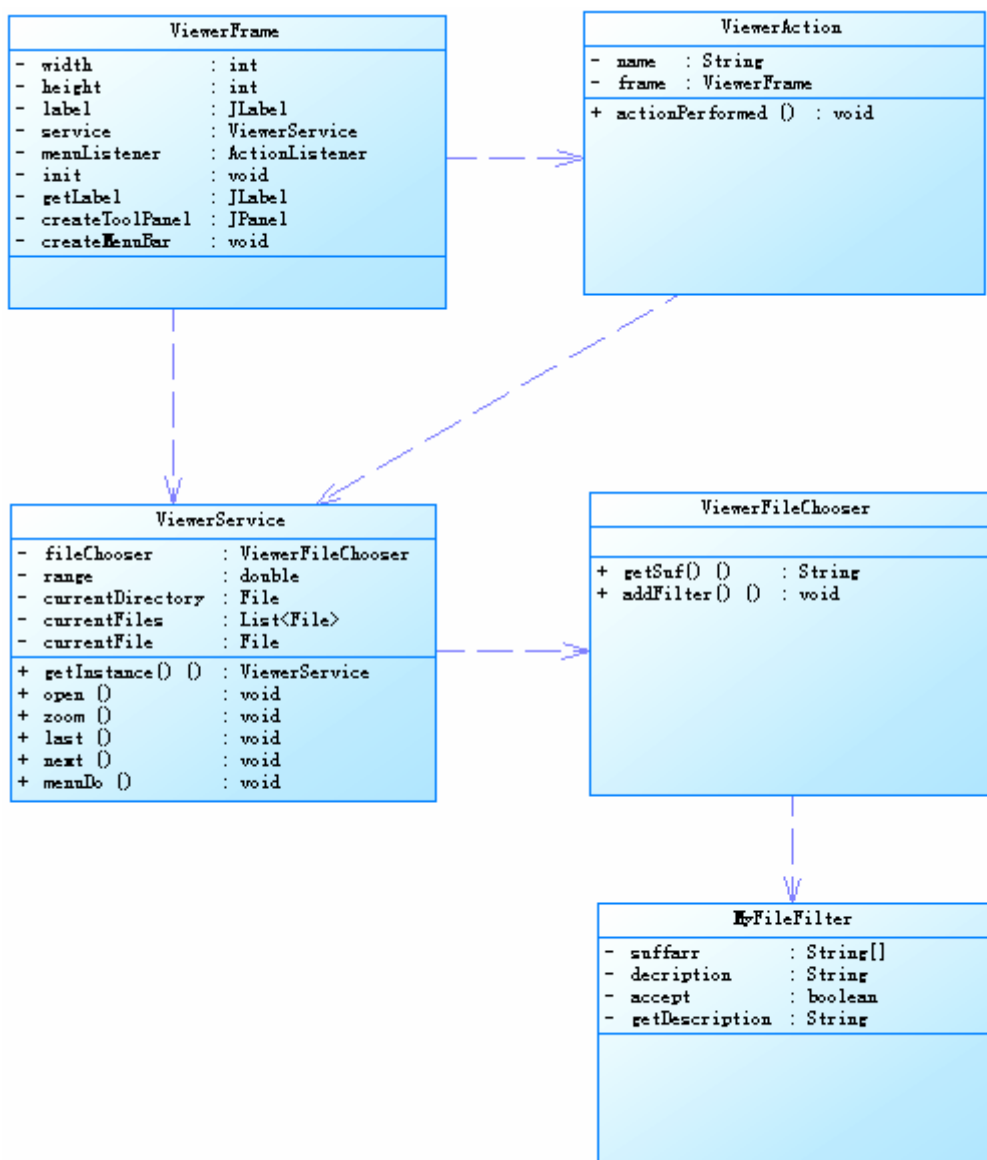


图 3.2 图片浏览器类图

本程序的功能较为简单，因此所涉及的对象也并不复杂，只有简单的五个对象。

### 3.2.1 文件过滤器

如果要使文件对话框实现文件过滤功能，就需要结合 `FileFilter` 类来进行文件操作，文件过滤器是 `FileFilter` 的一个继承，也是文件对话框的内部类，里面重写了 `FileFilter` 的 `accept` 与 `getDescription` 方法：

- ❑ `boolean accept( File f )`，判断文件是否属于图片类型。

- ❑ `String getDescription()`，获取过滤器的描述。

文件过滤器主要在用户打开图片时使用，当用户进行了图片选择后，就可以对用户所选择的文件进行验证。当用户打开文件选择时，我们就可以对所有的文件进行一次过滤，文件选择器中只可以选择我们所定义的图片文件，那么其他的文件将不会被显示。在本章中，文件过滤器是文件对话框类（`ViewerFileChooser`）的一个内部类（`MyFileFilter`）。

### 3.2.2 文件对话框

Java 文件对话框的实现比较简单，只要使用 `JFileChooser` 类并提供一个自己的构造器即可。这里的文件对话框对象是 `JFileChooser` 类的子类，目的是为了加入在 3.2.1 中定义的文件过滤器：

- ❑ `void addFilter()`，为这个文件对话框增加过滤器。

该对象中的 `addFilter` 方法主要用于向文件对话框加入文件过滤器，例如我们需要只显示 `.bmp` 的文件，那么可以在 `addFilter` 方法中使用以下代码实现：

```
this.addChoosableFileFilter(new MyFileFilter(new String[] { ".BMP" },  
"BMP (*.BMP)");
```

在文件对话框的 `addFilter` 方法加入以上的代码后，那么用户将不能看到 `.bmp` 的文件，并且在“文件类型”的下拉中也只能选择 `.bmp`，效果如图 3.3 所示。在本章中，文件对话框对应的是 `ViewerFileChooser` 类。

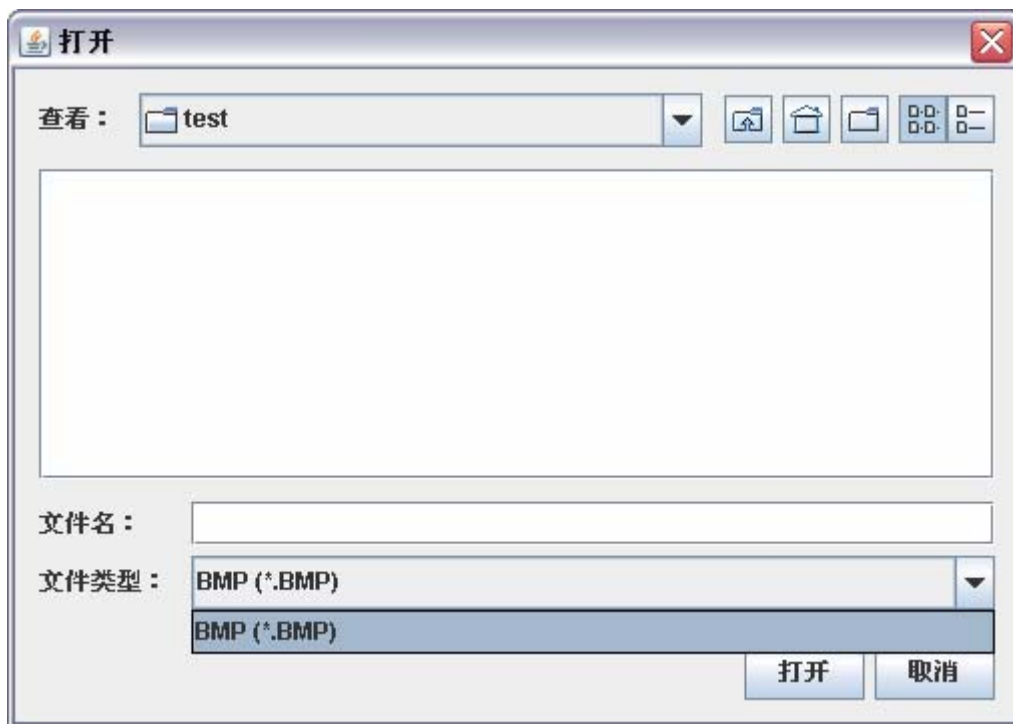


图 3.3 文件过滤器的作用

### 3.2.3 主界面类

我们建立一个界面类作为图片浏览器的主界面，该类包括图片显示区、菜单栏、工具栏，并为工具栏与菜单栏加上事件监听器，如下：

- ❑ `void init()`，初始化图片浏览器的界面。
- ❑ `JLabel getLabel()`，获取显示图片的 `JLabel`。
- ❑ `createToolPanel()`，创建放大、缩小、上一张、下一张等工具按钮。
- ❑ `void createMenuBar()`，创建文件、工具、帮助等菜单。

在这里需要注意的是，由于打开的图片大小并不能确定，因此图片显示区必须使用 `JScrollPane`。在本章中，主界面对应的是 `ViewerFrame` 类。

### 3.2.4 业务处理类 `ViewerService`

业务处理类主要是处理图片浏览器的大部分业务逻辑，包括打开图片、关闭浏览器、放大图片、缩小图片、浏览上一张图片、浏览下一张图片等功能，如下：

- ❑ `static ViewerService getInstance()`，获取 `ViewerService` 类的一个单态实例。
- ❑ `void open( ViewerFrame frame )`，弹出文件选择框，并读取被选择到的图片。
- ❑ `void zoom( ViewerFrame frame, boolean isEnlarge )`，对正在浏览到的图片做放大或者缩小操作，这里可能会丢失图片精度。
- ❑ `void last( ViewerFrame frame )`，浏览上一张图片。
- ❑ `next( ViewerFrame frame )`，浏览下一张图片。
- ❑ `void menuDo( ViewerFrame frame, String cmd )`，响应菜单的动作。

在本章中，这个业务处理类并不是无状态的 `Java` 对象，也就是意味着本章的业务处理类将人保存一些业务状态，这些业务状态包括：当前浏览的文件目录、文件目录的文件集合、图片放大或者缩小的比例等属性。由于我们这个是有状态的 `Java` 对象，那么就意味着，如果访问的是同一个实例，那么该对象的这些属性将会被所有的访问者共享，如果其中的一个访问者改变了其中一个或者多个属性，那么其他的访问者将会受到影响。当然，我们本章只是一个普通的图片浏览器，不存在多个用户使用同一个图片浏览器的情况。在本章中，业务处理类对应的是 `ViewerService` 类。

### 3.2.5 操作处理类

在本例中，由于用户可以执行的操作较少，因此，我们可以提供一个操作处理类来接收用户所有的操作，本例中的操作处理类是 `AbstractAction` 的一个子类，能用 `ImageIcon`（图标）来创建一个 `Action`，再用这个 `Action` 来创建按钮，点击按钮的时候，将调用此类的 `actionPerformed` 方法：

- ❑ `void actionPerformed((ActionEvent e)`，重写 `AbstractAction` 的方法，响应事件。

由于我们只有一个操作处理类，因此在实现 `actionPerformed` 方法时，我们就需要进行一系列的判断，让程序知道用户进行了何种操作，再调用业务处理类中的相应方法。

到此，图片浏览器的相关对象都已经建立，并且确定了我们需要实现哪些方法，我们在实现的过程中，如果发现可以对程序进行重构，那么也可以在重构的过程中，创建相关的类。

## 3.3 创建主界面

这个图片浏览器的界面排版比较简单，只有菜单（不需要排版）、工具栏、图片显示区，我们使用

`BorderLayout` 进行布局，把工具栏放在 `BorderLayout.NORTH`，把图片显示区放在 `BorderLayout.CENTER`。在本章中，由于打开图片的大小并不确定，因此我们需要使用一个 `JScrollPane` 来作为图片显示区域。

### 3.3.1 初始化界面（init()方法）

首先，设置 `JFrame` 窗口的标题，接下来初始化画图区域，初始化为白色，然后再获取 `PENCIL_TOOL`(铅笔)类型的 `Tool`，创建各种鼠标监听器，并在监听的执行方法中调用 `Tool` 的相应方法，最后获取左边工具栏面板、下面菜单栏面板、菜单，并把这些面板与画图获取加到 `JFrame` 中，见以下代码。

代码清单：code\viewer\src\org\crazyit\viewer\ViewerFrame.java

```
public void init() {
    //设置标题
    this.setTitle( "看图程序" );
    //设置大小
    this.setPreferredSize( new Dimension( width, height ) );
    //创建菜单
    createMenuBar();
    //创建工具栏
    JPanel toolBar = createToolPanel();
    //把工具栏和读图区加到 JFrame 里面
    this.add( toolBar, BorderLayout.NORTH );
    this.add( new JScrollPane(label), BorderLayout.CENTER );
    //设置为可见
    this.setVisible( true );
    this.pack();
}
```

首先是为 `JFrame` 设置标题，接下来设置大小，然后调用本类的 `createMenuBar()` 方法去创建菜单栏、调用 `createToolPanel()` 方法去创建工具栏，最后把菜单栏和图片显示区加到 `JFrame` 中（图片显示区只是一个 `JLabel`）。以上代码中的黑体部分，使用一个 `createToolPanel` 的方法来创建菜单，该方法将在下面章节中实现。

### 3.3.2 创建菜单栏

菜单栏，必须有事件响应，所以，先为菜单定义一个事件监听器，见以下代码。

代码清单：code\viewer\src\org\crazyit\viewer\ViewerFrame.java

```
//加给菜单的事件监听器
ActionListener menuListener = new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        service.menuDo( ImageFrame.this, e.getActionCommand() );
    }
};
```

这个事件监听器实现了 `ActionListener` 中的 `actionPerformed` 方法，是响应用户操作的方法，方法里面的 `service` 类就是我们的业务逻辑处理类 `ImageService` 的一个单态实例。有了这个事件监听器，就可以一次性创建出所有的菜单（用数组定义好菜单文字等东西的形式），见以下方法。

代码清单：code\viewer\src\org\crazyit\viewer\ViewerFrame.java

```
public void createMenuBar() {
    //创建一个 JMenuBar 放置菜单
```



```

JMenuBar menuBar = new JMenuBar();
//菜单文字数组，以下面的 menuItemArr 一一对应
String[] menuArr = { "文件(F)", "工具(T)", "帮助(H)" };
//菜单项文字数组
String[][] menuItemArr = {
    {"打开(O)", "-", "退出(X)"},
    {"放大(M)", "缩小(O)", "-", "上一个(X)", "下一个(P)"},
    {"帮助主题", "关于"}
};
//遍历 menuArr 与 menuItemArr 去创建菜单
for( int i = 0 ; i < menuArr.length ; i++ ) {
    //新建一个 JMenu 菜单
    JMenu menu = new JMenu( menuArr[i] );
    for( int j = 0 ; j < menuItemArr[i].length ; j++ ) {
        //如果 menuItemArr[i][j]等于 "-"
        if ( menuItemArr[i][j].equals( "-" ) ) {
            //设置菜单分隔
            menu.addSeparator();
        } else {
            //新建一个 JMenuItem 菜单项
            JMenuItem menuItem = new JMenuItem( menuItemArr[i][j] );
            menuItem.addActionListener( menuListener );
            //把菜单项加到 JMenu 菜单里面
            menu.add( menuItem );
        }
    }
    //把菜单加到 JMenuBar 上
    menuBar.add(menu);
}
//设置 JMenuBar
this.setJMenuBar( menuBar );
}

```

图片浏览器的菜单是这样的结构：

```

文件(F)
    打开(O)
    退出(X)
工具(T)
    放大(M)
    缩小(O)
        上一个(X)
        下一个(P)
帮助(H)
    帮助主题
    关于

```

从代码中可以看到，程序用两个数组把这两层菜单的文字保存了进去，两个数组一起遍历，每次都创建一个菜单项（`JMenuItem`），并为这个菜单项增加上前面定义的事件监听器，然后把这个菜单项加到 `JMenu` 中。每次遍历完 第一个数组，都把这个 `JMenu` 加到 `JMenuBar` 中。遍历完所有数组，就把这个 `JMenuBar` 加到 `JFrame` 里面，创建菜单的过程就完成了。

### 3.3.3 创建工具栏

这里的工具按钮，为了美观，想用图片的方式创建 `JButton`，这里就要用到 `AbstractAction`，也就是我们扩展的 `ViewerAction` 类，首先是用 `ViewerAction` 的 `ViewerAction(ImageIcon icon, String name, ViewerFrame frame)` 去创建一个 `ViewerAction`，参数里面的 `icon` 对象就是从本地路径中读了图标的图标类，然后以这个 `ViewerAction` 对象为参数去创建一个 `JButton`。见以下代码。

代码清单：code\viewer\src\org\crazyit\viewer\ViewerFrame.java

```
public JPanel createToolPanel() {  
    //创建一个 JPanel  
    JPanel panel = new JPanel();  
    //创建一个标题为"工具"的工具栏  
    JToolBar toolBar = new JToolBar( "工具" );  
    //设置为不可拖动  
    toolBar.setFloatable( false );  
    //设置布局方式  
    panel.setLayout( new FlowLayout( FlowLayout.LEFT ) );  
    //工具数组  
    String[] toolarr = { "open", "last", "next", "big", "small" };  
    for( int i = 0 ; i < toolarr.length ; i++ ) {  
        ViewerAction action = new ViewerAction(  
            new ImageIcon("img/" + toolarr[i] + ".gif")  
            , toolarr[i], this );  
        //以图标创建一个新的 button  
        JButton button = new JButton( action );  
        //把 button 加到工具栏中  
        toolBar.add(button);  
    }  
    panel.add( toolBar );  
    //返回  
    return panel;  
}
```

以上代码的黑体部分，我们使用了 `JButton` 来创建工具栏的图标，每一个 `JButton` 对象都使用 `ViewerAction` 作为构造参数，但是需要注意的是，各个 `JButton` 之间并不是共享一个 `ViewerAction` 的实例。创建完菜单与工具栏后，可以运行查看具体的效果，主界面的效果如图 3.4 所示。

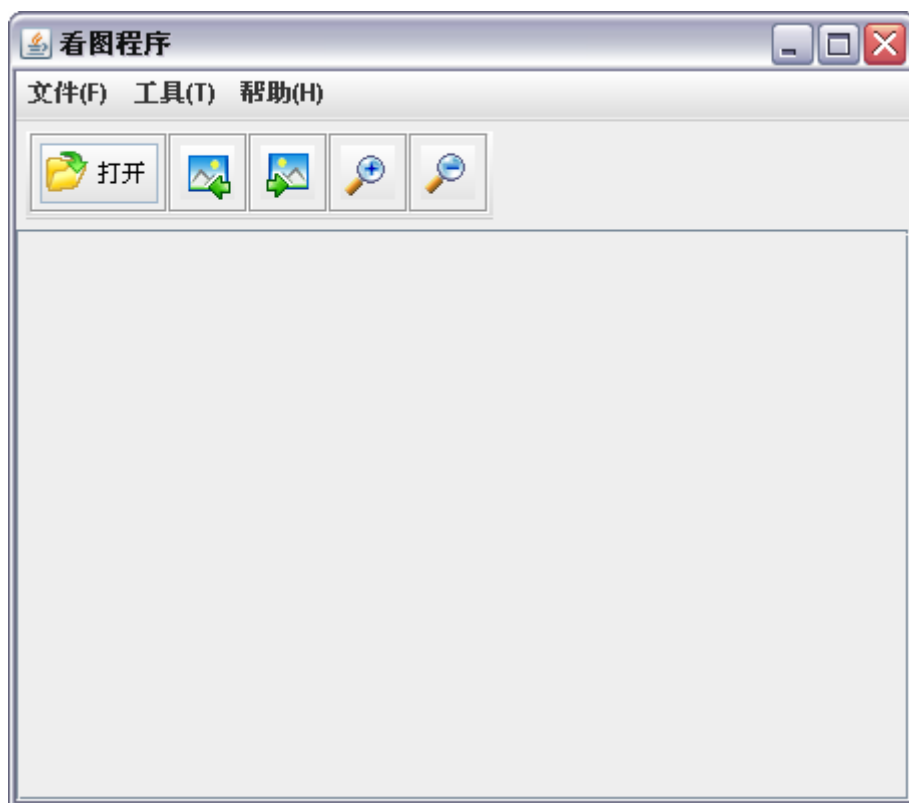


图 3.4 图片浏览器主界面

在本例中，图片浏览器的功能相对较为简单，因此界面也是较为简洁。如果想做更强大的图片浏览器，可以参考 ACESee 或者 Windows 图片浏览器等功能。

## 3.4 实现图片浏览的操作

**ViewerService** 类主要是处理图片浏览器的大部分业务逻辑，包括打开图片、关闭浏览器、放大图片、缩小图片、浏览上一张图片、浏览下一张图片等功能，在这里需要再做一次说明，**ViewerService** 是有状态的 Java 对象。

### 3.4.1 实现工具栏点击

我们在 3.2.5 中创建了一个 **ViewerAction** 的类，主要用于处理工具栏的点击事件，当用户点击了工具栏的某个操作时，就会执行 **ViewerAction** 的 **actionPerformed** 的方法。我们在 3.3.3 中创建工具栏时，使用了以下代码。

代码清单：code\viewer\src\org\crazyit\viewer\ViewerFrame.java

```
String[] toolarr = { "open", "last", "next", "big", "small" };
for( int i = 0 ; i < toolarr.length ; i++ ) {
    ViewerAction action = new ViewerAction(
        new ImageIcon("img/" + toolarr[i] + ".gif")
        , toolarr[i], this );
    //以图标创建一个新的 button
    JButton button = new JButton( action );
```

```
//把 button 加到工具栏中
toolBar.add(button);
}
```

以上代码中使用了“open”、“last”等字符串用来标识应该使用 `ViewerService` 的哪个方法，那么就意味着我们需要在 `actionPerformed` 方法中作出这些判断：

```
if (this.name.equals("open")) {
    //打开文件对话框
} else if (this.name.equals("last")) {
    //上一下图片
}
...
}
```

本章中只有 5 个 `Action`，就需要写 5 次的 `if...else`，对于这样的代码，我们在本书的第二章（仿 Windows 计算器）中已经出现，当前并没有提供任何解决方案，但是如果程序中出现如些之多的 `if...else`，那么我们就需要想办法去解决。接下来，创建一个 `Action` 的接口，提供一个 `execute` 的方法。

代码清单：code\viewer\src\org\crazyit\viewer\action\Action.java

```
public interface Action {
    /**
     * 具体执行的方法
     * @param service 图片浏览器的业务处理类
     * @param frame 主界面对象
     */
    void execute(ViewerService service, ViewerFrame frame);
}
```

编写了接口 `Action` 后，我们定义了一个 `execute` 的方法，那么，我们可以为该 `Action` 新建实现类，例如有一个打开文件对话框的 `Action`，那么我们就新建一个 `OpenAction`，该类实现 `Action` 接口。以下代码是 `OpenAction` 的具体的实现。

代码清单：code\viewer\src\org\crazyit\viewer\action\OpenAction.java

```
public void execute(ViewerService service, ViewerFrame frame) {
    //打开文件对话框
}
```

提供了这个 `OpenAction` 后，我们需要修改创建工具栏的代码，换一种方式创建工具栏。

代码清单：code\viewer\src\org\crazyit\viewer\ViewerFrame.java

```
// 工具数组
String[] toolarr = { "org.crazyit.viewer.action.OpenAction",
    "org.crazyit.viewer.action.LastAction",
    "org.crazyit.viewer.action.NextAction",
    "org.crazyit.viewer.action.BigAction",
    "org.crazyit.viewer.action.SmallAction" };
for (int i = 0; i < toolarr.length; i++) {
    ViewerAction action = new ViewerAction(new ImageIcon("img/"
        + toolarr[i] + ".gif"), toolarr[i], this);
    // 以图标创建一个新的 button
    JButton button = new JButton(action);
    // 把 button 加到工具栏中
    toolBar.add(button);
}
```

将原来的字符串更换为某个 `Action` 实现类的全限定类名，那么在构造 `ViewerAction` 的时候，就可以使用这个参数去创建具体的某个实现类。为 `ViewerAction` 编写一个工具方法，使用反射得到 `Action` 接口的某个实现类。

代码清单: code\viewer\src\org\crazyit\viewer\ViewerAction.java

```
private Action getAction(String actionName) {
    try {
        if (this.action == null) {
            //创建 Action 实例
            Action action = (Action)Class.forName(actionName).newInstance();
            this.action = action;
        }
        return this.action;
    } catch (Exception e) {
        return null;
    }
}
```

以上的黑体代码,使用了反射来创建一个实例,并且该实例在 `ViewerAction` 中只有一个实例,由于该方法在 `ViewerAction` 中,所以我们在构造 `ViewerAction` 的时候,将对应的处理类传入即可。得到具体的某个 `Action` 实现类后,在实现 `ViewerAction` 的时候,我们就可以不必使用那堆烦人的 `if...else` 了,直接通过以上的工具方法(`getAction`)得到相关的 `Action` 实现类,再调用 `Action` 的 `execute` 方法即可。

代码清单: code\viewer\src\org\crazyit\viewer\ViewerAction.java

```
public void actionPerformed(ActionEvent e) {
    ViewerService service = ViewerService.getInstance();
    Action action = getAction(this.actionName);
    //调用 Action 的 execute 方法
    action.execute(service, frame);
}
```

其实在本章中,我们并不需要如此复杂来实现,或许有些读者会觉得,编写多几个 `if...else` 可能比这样做更省事,但是,如果站在程序可扩展的角度看,当需要为图片浏览器添加行为时,我们就不必再修改 `ViewerAction`,我们这样做,无论添加或者减少多少个 `Action`,都不必去修改 `ViewerAction` 类,只需要去修改使用者(主界面对象)。对于一些简单的程序,我们可以使用 `if...else` 来解决,但是没有人知道程序将会有多复杂,因此笔者还是推崇使用其他方法来减少 `if...else` 或者尽量减低程序的耦合。

### 3.4.2 实现菜单的点击

我们为菜单增加了事件监听器,每次点击菜单时,都会先调用这个方法,由这个方法去决定做些什么类型的业务处理。在方法中,是根据菜单的文字去判断下步要调用的方法。

代码清单: code\viewer\src\org\crazyit\viewer\ViewerFrame.java

```
public void menuDo( ViewerFrame frame, String cmd ) {
    //打开
    if( cmd.equals("打开(O)") ) {
        open( frame );
    }
    //放大
    if( cmd.equals("放大(M)") ) {
        zoom( frame, true );
    }
    //缩小
    if( cmd.equals("缩小(S)") ) {
        zoom( frame, false );
    }
    //上一个
```

```
        if( cmd.equals("上一个(X)") ){
            last( frame );
        }
        //下一个
        if( cmd.equals("下一个(P)") ){
            next( frame );
        }
        //退出
        if( cmd.equals("退出(X)") ){
            System.exit( 0 );
        }
    }
}
```

在此，我们同样可以使用 3.4.1 中的方法来消除这一堆的 `if...else`，在这里不再详细描述。

### 3.4.3 打开图片

这个图片浏览器，打开一个图片文件之后，会把这个文件所有文件夹类的所有图片类型的的文件缓存起来，目的是为了不用每次都去搜索这个文件夹内的文件，也方便“上一张”和“下一张”的定位，缓存的文件都保存在本类的 `currentFiles` 中，`currentFiles` 是一个 `List<File>` 类型。

代码清单：code\viewer\src\org\crazyit\viewer\ViewerService.java

```
public void open( ViewerFrame frame ) {
    //如果选择打开
    if( fileChooser.showOpenDialog( frame )
        == ViewerFileChooser.APPROVE_OPTION ) {
        //给目前打开的文件赋值
        this.currentFile = fileChooser.getSelectedFile();
        //获取文件路径
        String name = this.currentFile.getPath();
        //获取目前文件夹
        File cd = fileChooser.getCurrentDirectory();
        //如果文件夹有改变
        if( cd != this.currentDirectory
            || this.currentDirectory == null ) {
            //或者 fileChooser 的所有 FileFilter
            FileFilter[] fileFilters = fileChooser
                .getChoosableFileFilters();
            File files[] = cd.listFiles();
            this.currentFiles = new ArrayList<File>();
            for( File file : files ) {
                for( FileFilter filter : fileFilters ) {
                    //如果是图片文件
                    if( filter.accept( file ) ) {
                        //把文件加到 currentFiles 中
                        this.currentFiles.add( file );
                    }
                }
            }
        }
    }
    ImageIcon icon = new ImageIcon( name );
}
```

```

        frame.getLabel().setIcon( icon );
    }
}

```

首先用 `ViewerFileChooser` 对象的 `showOpenDialog` 方法弹出一个文件选择框，在用户未选择图片之前，做其它操作的时候，这里就获取当前的文件路径与当前的文件夹。

如果 `currentDirectory`（当前文件夹）为空（证明是第一次打开文件）或者是 `currentDirectory` 不等于现在打开的文件夹，那么证明文件夹的路径有改变，就读取这个文件夹下面的所有文件。

在读取文件的过程中，先调用 `ViewerFileFilter` 中的 `getChoosableFileFilters()` 方法获取我们自定义的文件过滤器，如果读取到的文件类型属于当前的文件过滤器中允许的类型，就把这个文件加到 `currentFiles` 中缓存起来。

最后，用当前选择到的文件为参数新建一个 `ImageIcon` 对象，并调用 `ViewerFrame` 对象中 `JLabel` 对象的 `setIcon` 方法，把图片设置进去，就完成了显示图片的过程。

### 3.4.4 放大或者缩小图片

`Image` 中有一个叫 `getScaledInstance` 的方法，能根据宽度去按比例改变图片的大小。在这个缩放方法（`zoom`）中，用参数 `isEnlarge` 是代表放大或者缩小的。如果 `isEnlarge` 等于 `true`，就代表是放大，反之是缩小。

代码清单：code\viewer\src\org\crazyit\viewer\ViewerService.java

```

public void zoom( ViewerFrame frame, boolean isEnlarge ) {
    //获取放大或者缩小的乘比
    double enLargeRange = isEnlarge ? 1 + range : 1 - range;
    //获取目前的图片
    ImageIcon icon = (ImageIcon)frame.getLabel().getIcon();
    if( icon != null ) {
        int width = (int)(icon.getIconWidth() * enLargeRange);
        //获取改变大小后的图片
        ImageIcon newIcon = new ImageIcon( icon.getImage()
            .getScaledInstance( width,-1,Image.SCALE_DEFAULT) );
        //改变显示的图片
        frame.getLabel().setIcon( newIcon );
    }
}

```

首先是通过 `isEnlarge` 去得到缩放的比例（放大是大于 1，缩小是 0 与 1 之间），接下来从 `JLabel` 中用 `getIcon` 方法获的 `ImageIcon` 图片对象，如果这个对象不为空，就从这个对象中调用 `getIconWidth` 方法得到宽度，并用这个宽度和缩放比例相乘得到新的宽度。

用新的宽度为参数去调用 `getScaledInstance` 方法得到新的 `ImageIcon` 对象，最后又调用 `JLabel` 的 `setIcon` 方法把这图片设置到 `JLabel` 对象中去。

### 3.4.5 “上一张”、“下一张”图片

前面知道，`ViewerService` 中保存着当前打开的文件 `currentFile`，还有这个文件夹下面的所有图片文件 `currentFiles`，那么，读取“上一张”或者“下一张”图片就变的简单了，只要得到一个图片的索引，就能从 `currentFiles` 中取到图片。

这里是以读取上一张图片的方法为例子说明，读取下一张图片的实现是类似的。

代码清单：code\viewer\src\org\crazyit\viewer\ViewerService.java

```

public void last( ViewerFrame frame ) {
    //如果有打开包含图片的文件夹

```

```

        if( this.currentFiles != null && !this.currentFiles.isEmpty() ) {
            int index = this.currentFiles
                .indexOf( this.currentFile );
            //打开上一个
            if( index > 0 ) {
                File file = (File)this.currentFiles.get( index - 1);
                ImageIcon icon = new ImageIcon( file.getPath() );
                frame.getLabel().setIcon( icon );
                this.currentFile = file;
            }
        }
    }
}

```

如果 `currentFile` 与 `currentFiles` 都不为空（证明当前是有打开文件的），那就用 `currentFile` 从 `currentFiles` 中得到当前文件的索引，并把这个索引减 1，去获取上一个文件。

获取到上一个文件后，调用 `File` 类的 `getPath()` 方法得到文件的路径，然后以这个为参数来创建一个 `ImageIcon`，最后把它设置到 `JLabel` 中。

### 3.5 文件选择与过滤

使用 `JFileChooser` 创建文件对话框流程是先使用构造器创建一个 `JFileChooser` 对象，然后调用 `JFileChooser` 对象的 `showXXXDialog` 的方法显示文件对话框，如果需要对文件进行过滤，就需要调用 `addChoosableFileFilter(FileFilter filter)` 方法添加文件过滤器，见以下代码。

代码清单：code\viewer\src\org\crazyit\viewer\ViewerFileChooser.java

```

private void addFilter() {
    this.addChoosableFileFilter( new MyFileFilter(
        new String[]{ ".BMP", "BMP (*.BMP)" } );
    this.addChoosableFileFilter( new MyFileFilter(
        new String[]{ ".JPG", ".JPEG", ".JPE", ".JFIF",
            "JPEG (*.JPG;*.JPEG;*.JPE;*.JFIF)" },
        "JPEG (*.JPG;*.JPEG;*.JPE;*.JFIF)" );
    this.addChoosableFileFilter( new MyFileFilter(
        new String[]{ ".GIF", "GIF (*.GIF)" } );
    this.addChoosableFileFilter( new MyFileFilter(
        new String[]{ ".TIF", ".TIFF", "TIFF (*.TIF;*.TIFF)" } );
    this.addChoosableFileFilter( new MyFileFilter(
        new String[]{ ".PNG", "PNG (*.PNG)" } );
    this.addChoosableFileFilter( new MyFileFilter(
        new String[]{ ".ICO", "ICO (*.ICO)" } );
    this.addChoosableFileFilter( new MyFileFilter(
        new String[]{ ".BMP", ".JPG", ".JPEG", ".JPE", ".JFIF",
            ".GIF", ".TIF", ".TIFF", ".PNG", ".ICO",
            "所有图形文件" } );
}

```

这里是把 `bmp`，`jpg`，`gif` 等类型的文件过滤器都加到 `JFileChooser` 中，留意到这里是调用 `MyFileFilter( String[] suffarr, String decription )` 这个构造器去创建一个 `FileFilter`，第一个参数是后缀名，第二个参数是描述，见以下代码。

代码清单：code\viewer\src\org\crazyit\viewer\ViewerFileChooser.java

```

public MyFileFilter( String[] suffarr, String decription ) {
    super();
}

```



```
        this.suffarr = suffarr;  
        this.decription = decription;  
    }
```

**MyFileFilter** 继承了 **FileFilter**，我们这里重写它的 **accept** 方法，去定义过滤的规则，见以下代码。  
代码清单：code\viewer\src\org\crazyit\viewer\ViewerFileChooser.java

```
    public boolean accept( File f ) {  
        //如果文件的后缀名合法，返回 true  
        for ( String s : suffarr ) {  
            if ( f.getName().toUpperCase().endsWith( s ) ) {  
                return true;  
            }  
        }  
        //如果是目录，返回 true,或者返回 false  
        return f.isDirectory();  
    }
```

到此，我们整个图片浏览器已经实现，本章所涉及的内容较少，到现在可能运行程序查看具体的效果。

## 3.6 本章小结

本章通过图片浏览器的基本实现，向读者介绍了 **JFileChooser** 文件选择框与 **FileFilter** 的用法，使用了 **AbstractAction** 去创建按钮，并响应按钮事件。在代码中，使用 **List** 缓存的技巧简单去实现了“上一张”、“下一张”图片的功能，并让读者体会到可以怎样去操作一张图片，例如改变图片的大小等操作。在本章实现工具栏点击时，我们使用了 **Java** 的反射来创建具体具体的某个工具栏 **Action** 类，我们本章中初步使用了 **Java** 的反射，在以后的章节中，我们会更多的使用各种 **Java** 技术，目的是为了减低程序的耦合，编写更多可扩展的程序。

## 第 4 章 桌面弹球

### 4.1 桌面弹球概述

桌面弹球是游戏中常见的游戏，从以前的掌上游戏机到如今的手机游戏，都是一个十分经典的游戏。玩家控制一个可以左右移动的挡板去改变运动中小球的移动方向，目的是用小球消除游戏屏幕中的所有障碍物到达下一关，在障碍物被消除的过程中，可能会产生一些能改变挡板或者小球状态的物品，例如：挡板变长、变短，小球威力加强等等。本章主要介绍如何实现一个简单的弹球游戏，让读者了解“动画”的实现原理。

在本章中，将介绍与使用 Java 的绘图功能，使用到 JPanel 的 `paint(Graphics g)` 方法去绘图，绘图主要是依靠这个方法中的 `Graphics` 类型的参数，将使用 Java 中的 `Timer` 去重复绘图，产生动画效果，桌面弹球游戏的效果如图 4.1 所示。

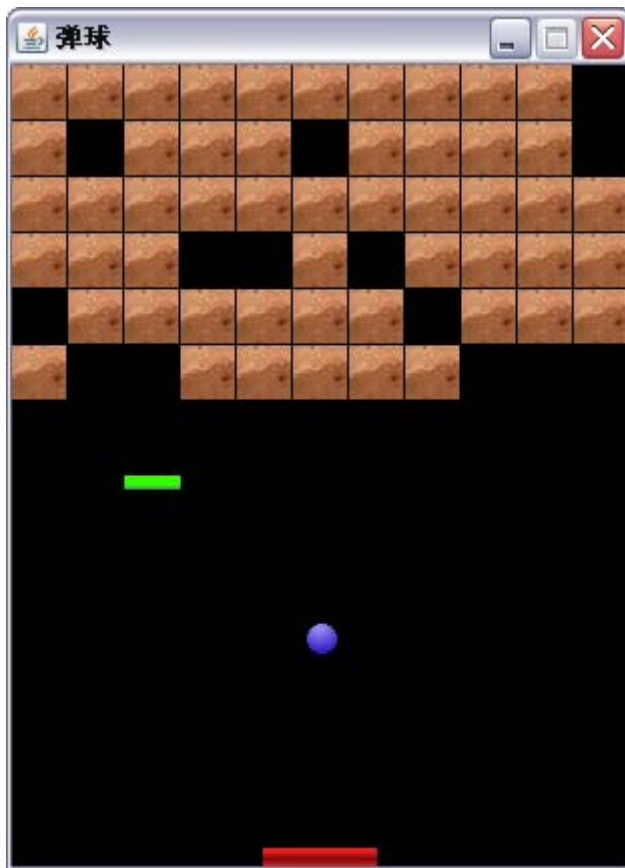


图 4.1 桌面弹球

#### 4.1.1 动画原理

简单地来说，动画是利用人的视觉暂留的生理特性，实现出来的一种假象，只要每隔一段时间（这个时间少于人的视觉暂留时间）就重新绘制一幅状态改变的图片，就能造成这种“动”的假象。我们在

程序中不断的进行绘画（使用 `repaint` 方法），对程序来讲，只需要在短时间内进行多次的绘画，并且每次绘画都需要改变绘画的相关值，就可以达到“动画”的效果。

### 4.1.2 小球反弹的方向

在本章实现的过程中，我们会设置小球于对称的方式，并出现少许偏移的方式反弹，如图 4.2 所示。让小球反弹出现少许偏移是为了让游戏增加点不确定性，增加游戏的趣味性。我们需要在编写游戏前确定这些小细节，这样在开发的过程中，我们就可以按照这些小细节去逐步实现我们的程序。

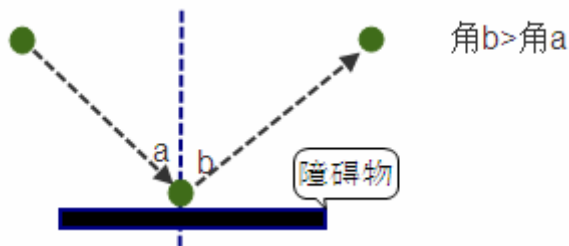


图 4.2 小球的反弹

## 4.2 流程描述

玩家使用左右方向键开始游戏与控制挡板，在未消除完所有的障碍物或者挡板没有挡住向下移动的小球之前，会一直处于游戏状态，在这个状态中，小球会一直处于直线运动或者改变方向，当小球消除掉障碍物的时候，有机率产生一些物品，产生的物品会直线向下移动，用挡板接住物品后，物品的特殊效果会生效。如果消除了所有的障碍物，就判断玩家为赢，如果挡板没有接住向下移动的小球，就判断玩家为输。具体的游戏流程如图 4.3 所示。

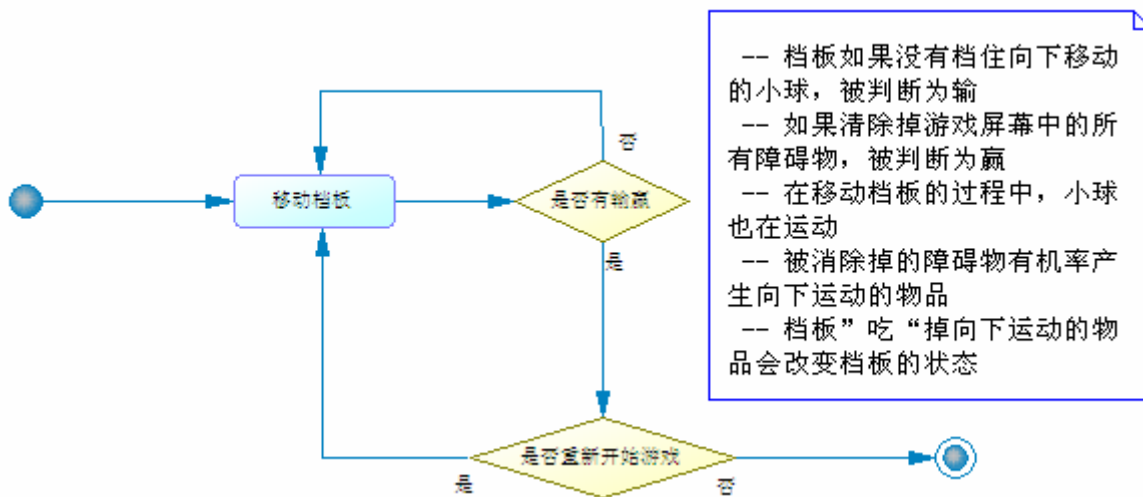


图 4.3 游戏流程

游戏中并不涉及复杂的流程，只需要处理游戏的输赢即可，因此在实现的过程中，关键是如何确定游戏输赢的标准（挡栏没有挡住小球）。

## 4.3 创建游戏对象

在这个游戏中，有挡板，小球，砖块（障碍物），道具等物品，这些物品都有共同的特性，有属于自己的  $x$  与  $y$  坐标属性，有图片属性，有速度属性，所以，在这时在，设计一个基类 **BallComponent** 包含这些属性与相关的方法，让其子类继承。继承此类的子类有 **Stick** 类（用于定义挡板的的行为于属性），**Ball** 类（控制小球的移动与其它动作），**Brick** 类（砖块类），**Magic** 类（道具抽象类，此类中有一个用于使道具功能实现的抽象方法，供其子类实现）。道具类的子类有 **LongMagic** 与 **ShortMagic**，作用是使 **Stick** 的长度变长或者变短。在平时的开发中，如果发现多个对象之间有一些共同的特性或者行为，并且觉得可以使用这些特性或者行为构成一个对象，那么可以建立一个新的对象作为这些对象的父类。如果该父类中某些方法并不需要由父类实现，我们可以将父类做成抽象类，并将这些方法变成抽象的。

确定了我们游戏中的所涉及的对象后，我们还需要一个 **BallFrame** 类去创建一个画板，用于绘制图片，此类还完成界面的初始化，监听用户的键盘，而与游戏相关的业务逻辑（判断输赢或者球的运动），我们放到 **BallService** 类中去处理，本章类的关系如图 4.4 所示。

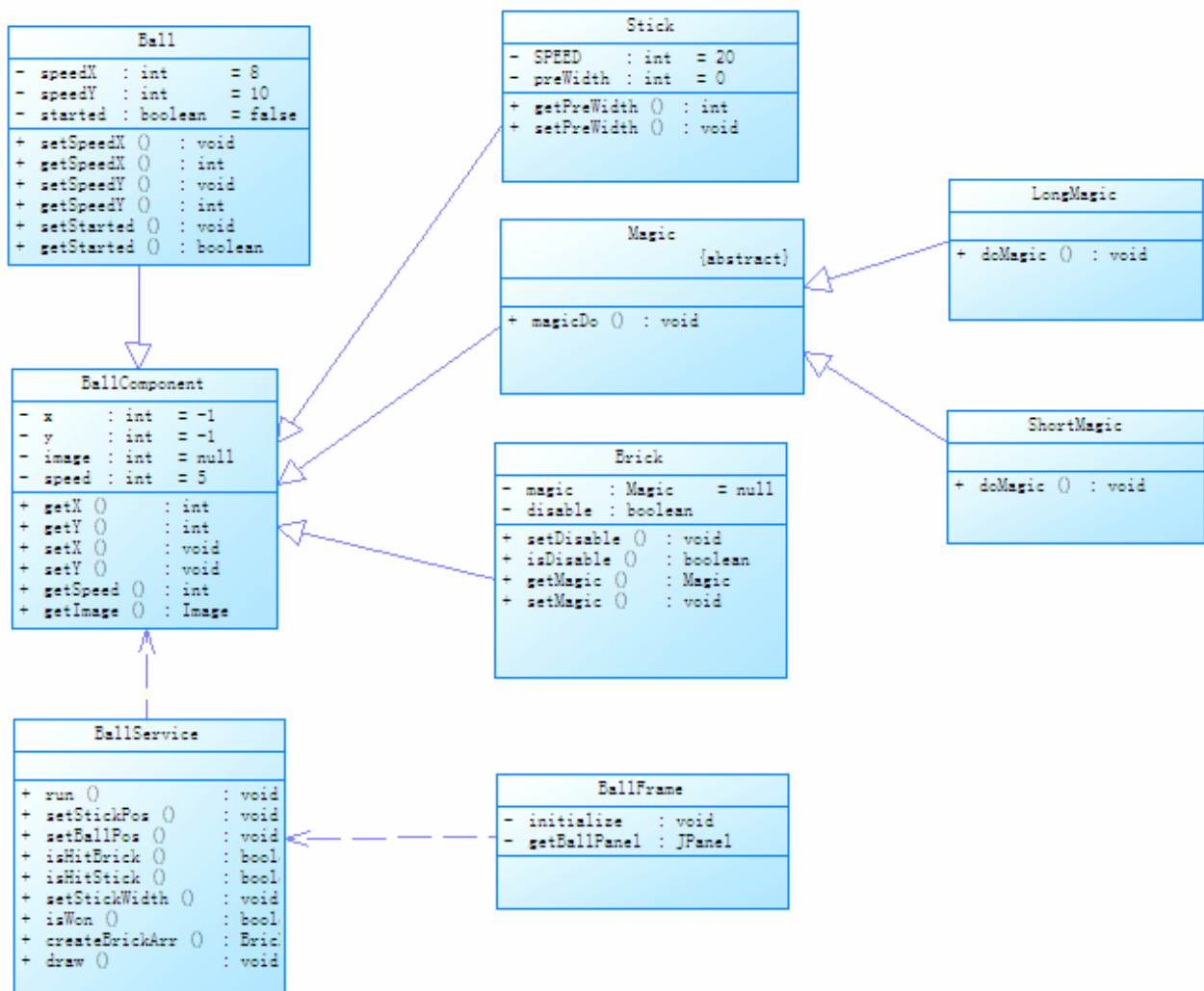


图 4.4 桌面弹球类图

笔者在这里提供了本章的类图，是为了让读者可以更清晰的了解本小程序的结构，但在实现开发的过程中，我们可以根据实际情况，加入或者改变各个类的关系或者程序的结构，但最终都是为降低程序的耦合、提高内聚、编写出优秀的代码。

### 4.3.1 基类BallComponent

**BallComponent**，做为 **Brick**（砖块）类、**Magic**（道具）类、**Stick**（挡板）类、**Ball**（小球）类的父类，定义了这些子类共有的属性与方法，属性有：**x** 坐标，初始值为-1；**y** 坐标，初始值为-1；图片 **image**，初始值为 **null**；速度 **speed**，初始值为 5。根据不同的需要，提供以下三个构造方法：

- ❑ **BallComponent( String path )**，**path** 是图片的路径，用图片的路径来构造一个 **BallComponent**，在此构造方法中，将根据路径去读取图片，再设置对象中 **image** 属性。
- ❑ **BallComponent( int panelWidth , int panelHeight, String path )**，以 **panelWidth**，**panelHeight**，与 **path** 去构造一个 **Component**。
- ❑ **BallComponent( String path , int x , int y )**，以 **x** 坐标，**y** 坐标和 **path** 去构造一个 **BallComponent**。

除去这些构造方法，此类提供了这些属性的 **setter** 与 **getter** 方法，用于获取对象的坐标与图片，或者改变对象的坐标位置与图片属性。如果我们在编码的过程中发现有一些共同的属性或者方法，我们可以将这些放到这个基类中。

创建 **BallComponent** 的时候，我们可以将这个类变成抽象类，即使它没有任何的抽象方法，这样做的目的是，在我们的桌面弹球游戏中，该类并不是具体存在的某一个对象，而是我们将一些公用的属性或者方法存放到该类中，因此它在游戏中并不代表某个具体的对象。将该类创建为抽象类，我们就可以提供（如果需要的话）一些抽象方法让子类去实现，并且可以在父类中调用这些抽象方法。

### 4.3.2 砖块类（Brick）

此类是 **BallComponent** 的一个子类，提供一个 **Brick(String path, int type, int x, int y)** 构造器，其中 **path**、**x** 与 **y** 参数用于调用父类的构造器，**type** 是代表砖块的类型：1 代表此砖块里面有 **LongMagic** 类型的道具；2 代表此砖块里面有 **ShortMagic** 类型的道具；其它代表此砖块里面没有道具。另外，本类增加了 **magic** 与 **disable** 属性，**magic** 代表此砖块中所包含的道具，初始值为 **null**，**disable** 是用来标志 **Brick** 的状态，如果 **disable** 为 **true**，则表明此砖块已经不可用，不会再显示。并提供这两个属性相关的以下方法：

- ❑ **void setMagic( Magic magic )**，设置道具。
- ❑ **Magic getMagic()**，获取道具。
- ❑ **boolean isDisable()**，用来判断此类是否有效。
- ❑ **void setDisable( boolean disable )**，停用或者启用此类，**disable** 的值为 **true** 或者 **false**。

确定了一个砖块由一个 **Brick** 对象来表示后，在界面中，我们可以提供一个 **Brick** 的二维数组，来表示界面中所有的砖块，实现原理与控制台五子棋中的棋盘一样，但是在本章中，二维数组的每一个元素并不是字符串，而是具体的某个 **Brick** 对象，在以后的章节中，当遇到需要在界面中绘画某些图片的时候，我们都可以建立一个二维数组，将相应的对象放置到该数组中，当界面进行绘画的时候，就可以将这个二维数组“画”出来。

### 4.3.3 道具类及其子类（Magic）

**Magic** 类是一个道具类，在游戏中表现包含在砖块中，是 **BallComponent** 的一个抽象子类，此类提供一个 **Magic( String path, int x , int y )** 构造器去调用父类的构造器，并提供一个抽象的方法 **magicDo( Stick stick )**，此抽象方法是实现道具的效果功能，用于给其子类实现，现在实现的子类的 **LongMagic** 类和 **ShortMagic** 类，两个子类的 **magicDo** 方法中分别实现使挡板变长与变短的功能。

- ❑ **abstract void magicDo( Stick stick )**，道具的功能，给其子类实现。

在本例中，挡板是可以变长或者变短的，而使挡板变长或者变短的方式是通过道具来实现，因此可以将道具抽象成变长的道具或者变短的道具，而它们都需要做同一件事，就是改变挡板的展现形式。为

了程序的可扩展性，我们在这里将一个道具变为一个抽象类（**Magic**），当我们需要有其他形式的道具的时候，就可以为该类添加子类，并提供不同的实现。当然，这里只提供一个 **Stick** 的参数可能并不够，如果以后游戏中出现另外一种道具，会改变球的速度（变快或者变慢），那么我们就需要为该抽象类提供更多的参数。

#### 4.3.4 挡板类（Stick）

同样，**Stick** 类也是 **BallComponet** 的子类，用来代表游戏中的挡板，由于挡板只有左右移动的，所以，此类中只定义了挡板 **x** 方向的移动速度 **SPEED**，还有定义挡板的初始长度 **preWidth**，并提供此方法的 **setter** 与 **getter** 方法，如下：

- **void setPreWidth( int preWidth )**，设置初始长度。
- **int getPreWidth()**，获取初始长度。

由于该类继承于 **BallComponet** 类，因此只需要提供一个构造器即可。在本例中，挡板是可以变长或者变短的，并且在建立道具抽象类的时候，已经定义了一个 **magicDo** 的方法，该方法的参数就是一个挡板对象，所以挡板类必须包括长度的属性，这样，在实现道具类的时候，就可以通过改变挡板类的长度来实现本例中所需要实现的长短挡板功能。在 **Stick** 类中并不需要关心挡板的图片、位置与大小，这些属性已经在 **BallComponet** 中体现。

#### 4.3.5 小球类（Ball）

**Ball** 类也是 **BallComponet** 的子类，由于小球在游戏面板中运动的时候除了横竖方向，还有各种角度的斜方向，所以我们将小球的速度分解成横向速度与竖向速度（**speedX** 与 **speedY**），游戏未开始前，小球是处于静止状态，所以用一个 **started** 属性来标志小球是否已经开始运动。游戏结束后，小球也是处于静止状态，但不能再移动，同样，用一个 **stop** 属性来标志小球是否能再移动。除了定义这些属性，还为这些属性提供相应的 **setter** 与 **getter** 方法，如下：

- **setSpeedX( int speed )**，设置小球的横向速度。
- **setSpeedY( int speed )**，设置小球的竖向速度。
- **boolean isStarted()**，小球是否已经在运动。
- **void setStarted( boolean b )**，把小球状态设置为运动或者静止。
- **int getSpeedX()**，获取小球的横向速度。
- **int getSpeedY()**，获取小球的竖向速度。

在本例中，小球对象只保存一些相关的属性，例如横向速度与纵向速度（图片、位置与大小在父类中体现），如果需要改变小球的速度，可以调用相关的 **setter** 方法来进行，但是我们需要知道由哪些对象来改变小球的相关属性，我们在前面的章节中提到，提供一个业务类进负责处理游戏的相关逻辑，因此，业务类就需要维护一个小球的对象，来控制小球的运动或者其他行为。在这里，小球对象可以单纯的看作一个简单的对象，并不负责处理任何的行为，这可以看作我们一般所说的贫血模式，对象并不负责处理任何的业务逻辑。如果需要将该小球对象编写成为充血模式，可以为小球对象提供一些与之相关的行为，例如小球会运动，我们可以为 **Ball** 类加入一个 **run** 的方法，表示球的运动，例如小球会停止运动（在游戏结束或者开始时），我们就可以为 **Ball** 类添加一个 **stopRun** 的方法，总之，如果需要做到充血模式，可以将所有与小球相关的方法加入到 **Ball** 中。

#### 4.3.6 业务处理类（BallService）

**BallService** 处理了这个游戏中的大部分业务功能，包括开始游戏、小球移动、道具移动、挡板移动、测试小球与挡板是否有碰撞或者挡板和其它元素有碰撞、设置挡板的长度、判断用户是否通关、初

始化砖块的排列与道具、画图等功能。这些功能的实现都有对应的方法，如下：

- ❑ `void run()`，小球进行运动。
- ❑ `void setStickPos( KeyEvent ke )`，改变挡板的坐标位置。
- ❑ `setBallPos()`，改变小球的坐标位置。
- ❑ `boolean isHitBrick( Brick brick )`，测试小球与砖块是否有碰撞，参数 `brick` 是指砖块。
- ❑ `isHitStick( BallComponent image )`，测试某元素与挡板是否有碰撞。
- ❑ `void setMagicPos()`，改变道具的坐标位置。
- ❑ `void setStickWidth( Magic magic )`，根据道具（`magic`）的类型去设置改变挡板的长度。
- ❑ `boolean isWon()`，判断玩家是否已经过关。
- ❑ `Brick[][] createBrickArr( String path, int xSize, int ySize )`，创建砖块，返回一个 `Brick` 类型的数组，参数 `path` 是指砖块的图片，`xSize` 与 `ySize` 是数组的长度。
- ❑ `void draw( Graphics g )`，画图，方法中是使用 `Graphics` 对象 `g` 去画图。

当游戏开始时，程序中需要不停的调用 `run` 方法，让小球进行运动，当然，小球进行运动的前提是 `Ball` 的 `isStarted` 方法返回 `true`，即游戏已经开始，`run` 方法的主要功能就是调小球的位置。我们需要在游戏中通过上、下、左、右的键来控制挡板的位置，因此就需要提供一个 `setStickPos` 的方法来改变挡板的位置。在本章的程序中，`BallService` 处理所有的相关逻辑，例如判断小球在运动的过程中是否越界、游戏是否胜利等。在例中 `BallService` 处理了大部分的游戏逻辑，当然，我们也可将这些逻辑放到相关的类中（即前面提到的充血模式），例如道具的下落、挡板的移动等。

#### 4.3.7 主界面类（BallFrame）

`BallFrame` 是创建一个 `JFrame` 主界面，设置主界面的标题、长与宽、画板等属性，并且为增加键盘事件监听器以及创立一个 `Timer` 每隔一小段时间去刷新画板，主要有初始化界面与或者画板两个方法，如下：

- ❑ `void initialize() throws IOException`，此方法抛出 `IO` 异常，初始化界面。
- ❑ `BallPanel getBallPanel()`，获取一个 `BallPanel` 类型的 `JPanel` 去充当画板，`BallPanel` 是这个类中的一个内部类。

我们使用了 `BallService` 类来处理大部分的游戏逻辑，主界面类中几乎不包括任何的逻辑处理，该类维护一个 `BallService` 的对象，得到界面中相关对象的信息后，可以调用 `BallService` 中的方法进行处理，并根据返回的信息来改变界面。例如小球的运动，我们可以调用 `BallService` 的 `run` 方法，再调用 `BallService` 的 `draw` 方法将小球的图片“画”到界面中。

到此，本章中所有的对象都已经创建并确定了它们的行为，在建立道具类（`Magic`）的时候，我们将一个道具抽象为一个 `Magic` 对象，该类可以有多个实现，在使用 `Magic` 对象的时候，我们可以利用面向对象的多态特性，使用 `Magic` 的 `magicDo` 方法来进行“道具的使用”，在这个过程中，我们并不需要去关心道具具体的实现。在创建游戏各个对象的过程中，我们将处理逻辑的方法放置到一个业务类中，从一定程度上讲，减少了代码之间的耦合，并遵循了单一职责的原则。

### 4.4 主界面实现

在这个桌面弹球游戏中，游戏中的所有元素都是用 `Graphics` 对象画出来的，所以，我们的主界面应该是一个只设置了窗口标题还有颜色等基本属性的 `JFrame`，在这个 `JFrame` 中，我们只需要提供一个 `JPanel` 对象即可，因为游戏的界面并没有多复杂的布局与界面交互。当我们实现游戏的一些相关逻辑的时候（球的运动、道具的下落等），我们可以调用 `JPanel` 的 `repaint` 方法将 `JPanel` 进行重绘。

### 4.4.1 初始化界面（initialize()方法）

首先，设置 JFrame 窗口的标题、背景颜色与是否可以改变大小，然后获取 JPanel 对象，最后把 JPanel 画板加到 JFrame 中，见以下代码。

代码清单：code\ball\src\org\crazyit\ball\BallFrame.java

```
public void initialize() throws IOException {
    //设置窗口的标题
    this.setTitle("弹球");
    //设置为不可改变大小
    this.setResizable( false );
    //设置背景为黑色
    this.setBackground( Color.BLACK );
    //获取画板
    ballPanel = getBallPanel();
    //把画板加到 JFrame
    this.add( ballPanel );
}
```

看加粗的一行代码 **ballPanel = getBallPanel()** 是调用本类中的 **getBallPanel()** 方法去获取一个 BallPanel 对象，BallPanel 是本类的一个内部类，并且继承 JPanel，见以下代码。

代码清单：code\ball\src\org\crazyit\ball\BallFrame.java

```
//定义一个 JPanel 内部类来完成画图功能
public class BallPanel extends JPanel {
    /**
     * 重写 void paint( Graphics g )方法
     *
     * @param g Graphics
     * @return void
     */
    public void paint( Graphics g ) {
        //可以调用 BallService 的 draw 方法进行绘制
    }
}
```

而获取这个 BallPanel 实现是在 BallPanel getBallPanel 方法中，此类保证这个 Panel 是单态的，每次只有一个 BallPanel 对象，见以下代码。

代码清单：code\ball\src\org\crazyit\ball\BallFrame.java

```
public BallPanel getBallPanel() {
    if ( ballPanel == null ) {
        //新建一个画板
        ballPanel = new BallPanel();
        //设置画板的大小
        ballPanel.setPreferredSize(
            new Dimension( BALLPANEL_WIDTH, BALLPANEL_HEIGHT ) );
    }
    return ballPanel;
}
```

在这里需要注意的是，我们需要在 BallFrame 中维护一个 BallPanel 的对象，然后通过 getBallPanel 的方法来获得 BallPanel 的实例，由于 BallPanel 并不需要每次去创建，所以我们可以将 BallPanel 变成单态的。在众多的设计模式中，有一种叫做单态模式。如果遇到一些对象并不需要多次创建或者创建这些对象将会严重消耗系统资源，那么我们可以考虑将该对象写成单态的。



### 4.4.2 单态模式简介

单态模式也可以叫单例模式，该模式保证一个类有且仅有一个实例，并为外界提供一个访问，让外界可以通过这个访问点来访问该类的唯一实例。在我们平时开发的过程中，会遇到一些不需要多次创建的对象，例如 JDBC 的 **Connection** 对象，那么我们就可以利用单态模式来创建这些对象。例如单态模式，系统可以不必多次创建该对象的实例，外界使用的时候可以使用同一个实例，因此在一定程度上减低了系统在创建对象时的开销。

为一个类实现单态模式，需要为该类提供一个私有的构造器，再提供一个可以获取该类实现的方法（为外界提供唯一的访问点），私有构造器是为了不让外界去使用 **new** 关键字来创建该类的实现，如果外键可以使用 **new** 关键字来创建该类的实例，那么就意味着该类将不会是单态，有可能外界多次通过 **new** 关键字来创建，这就无法保证该对象的实例的唯一性。

### 4.4.3 运行效果

编写了 **BallFrame** 的初始化代码后，我们可以运行具体查看相关的游戏效果。编写创建 **BallFrame** 的代码：

```
BallFrame ballFrame = new BallFrame();  
ballFrame.pack();  
ballFrame.setVisible(true);  
ballFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

当前程序的效果如图 4.5 所示。

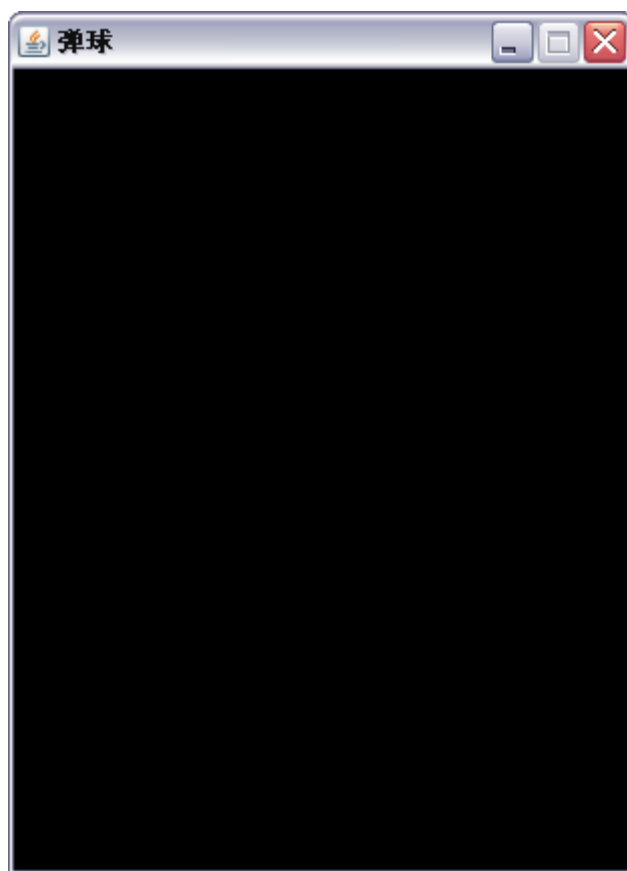


图 4.5 初始化游戏时的界面

注：我们当前并没有对 BallService 中的 draw 方法作任何的实现，我们实现了 BallService 的 draw 方法后，就可以将 BallPanel 中的 paint 方法加入 BallService.draw。

#### 4.4.4 监听器与Timer

javax.swing.Timer 可以设定每隔一个时间周期就重复执行某个 task，类似于 Window 系统的计划任务或者 Linux 系统的 cronjob，并用 start() 方法去启用 Timer。在这个弹球游戏中，我们只有键盘操作，所以只监听键盘的操作，用一个 KeyListener 去监听键盘的动作，请看以下代码。

代码清单：code\ball\src\org\crazyit\ball\BallFrame.java

```
//定义每 0.1 秒执行一次监听器
ActionListener task = new ActionListener(){
    public void actionPerformed( ActionEvent e ) {
        //开始改变位置
        service.run();
        //刷新画板
        ballPanel.repaint();
    }
};
//如果 timer 不为空，调用 timer 的 restart 方法
if( timer != null ) {
    //重新开始 timer
    timer.restart();
} else {
    //新建一个 timer
    timer = new Timer( 100, task );
    //开始 timer
    timer.start();
}
//增加事件监听器
KeyListener[] klarr = this.getKeyListeners();
if( klarr.length == 0 ) {
    //定义键盘监听适配器
    KeyListener keyAdapter = new KeyAdapter(){
        public void keyPressed( KeyEvent ke ) {
            //改变挡板的坐标
            service.setStickPos( ke );
        }
    };
    this.addKeyListener( keyAdapter );
}
```

首先，建立一个 ActionListener 对象做为 Timer 的 task，这个 task 主要是处理游戏中各个组件位置的改变以及 repaint 画板，这个 task 每 100 毫秒执行一次，即每隔一百毫秒小球（或者其他组件）会执行一次运动。如果此类的属性 timer 为空，就以 ActionListener 对象为参数去创建一个每 100 毫秒执行一次的 Timer，并用调用 start() 方法启动 Timer，如果 timer 不为空，直接调用 restart() 方法启动 timer。在这里我们需要明白的是，第一次进行游戏时，timer 为 null，就需要进行创建，当进行第二次游戏的时候，timer 非空，由于游戏停止（胜利或者失败），因此需要调用 restart 方法重新启动。由于我们是在 BallService 控制游戏的，也就意味着进行第二次游戏的时候，就需要再次调用 BallFrame 的 initialize 方法初始化游戏。

接下来再增加事件监听器,先使用 `JFrame` 的 `getKeyListeners()` 方法获取本窗口的 `KeyListener` 数组,如果这个数组的长度为空,说明本窗口并没有添加到任何 `KeyListener`,所以就创建一个 `KeyAdapter` (为 `JFrame` 创建一个键盘监听器)并重写 `KeyAdapter` 类的 `void keyPressed(KeyEvent ke)` 方法,这个方法用来监听键盘的按键是否有按下,如果有的话,就需要调用 `BallService` 的 `setStickPos` 方法。当我们要实现 `setStickPos` 方法的时候,就需要设置小球为运动状态,启动弹球游戏就意味着小球开始进行运动。当我们在游戏中按下左右键的时候,同时需要移动挡板,启动游戏后,我们并不需要关心小球的移动,仅仅设置小球的运动状态,换言之,`setStickPos` 方法只是处理挡板的移动,小球的运动让 `BallService` 的 `run` 处理 (`run` 方法 100 毫秒执行一次)。

## 4.5 挡板、小球、砖块、道具

在这个设计中,挡板、小球、砖块与砖块中所包含的道具都有一个共同的父类 `BallComponent`,可以使用父类的 `setX` 与 `setY` 方法设置坐标,也可以使用 `getX` 与 `getY` 方法获取坐标,还可以使用 `getImage` 方法获取图片,并且父类根据不同的情况提供了几个不同的构造器,

### 4.5.1 挡板 (Stick类)

此类提供一个以画板的宽、高和挡板的图片路径为参数的构造器,见以下代码。

代码清单: `code\ball\src\org\crazyit\ball\Stick.java`

```
public Stick( int panelWidth , int panelHeight, String path ) throws IOException {
    //调用父构造器
    super( panelWidth, panelHeight, path );
    //设置 y 坐标
    this.setY( panelHeight - super.getImage().getHeight( null ) );
    //设置原本的长度
    this.preWidth = super.getImage().getWidth( null );
}
```

首先调用父类的 `BallComponent(int x, int y, String path)` 构造器,把此对象的 `x` 坐标设置到画板中间的位置,并且使用 `javax.imageio.ImageIO` 的 `read` 方法去读取磁盘中的图片文件。接下来把 `y` 坐标设置到画板的底部,再根据读取出来的图片的宽度去设置 `Stick` 对象的初始长度属性。在从磁盘读取图片的过程是一个 `IO` 操作,所以会抛出 `IOException`,见以下代码。

代码清单: `code\ball\src\org\crazyit\ball\BallComponent.java`

```
public BallComponent( int panelWidth , int panelHeight, String path ) throws IOException {
    super();
    //读取图片
    this.image = ImageIO.read( new File( path ) );
    //设置 x 坐标
    this.x = (int)( ( panelWidth - image.getWidth( null ) ) / 2 );
}
```

由于挡板的长度可能会改变,所以 `Stick` 类有的个 `int` 类型的 `preWidth` 属性,代表挡板的长度,并定义一个 `final int` 类型的 `SPEED` 属性,代表挡板的移动速度,每次移动,`x` 坐标都会向左或者向右移动 `SPEED` 个坐标位置,需要为 `preWidth` 属性提供 `setter` 与 `getter` 方法。

实现了挡板类后,我们可以实现 `BallService` 的 `draw` 方法,先将挡板“画”到 `BallPanel` 中,并在 `BallPanel` 中调用 `BallService` 的 `draw` 方法,以下是 `BallService` 的 `draw` 方法的部分实现。

代码清单: `code\ball\src\org\crazyit\ball\BallService.java`

```
// 如果赢了
if (isWon()) {
```

```
// 绘制赢的图片
g.drawImage(won.getImage(), won.getX(), won.getY(), width,
            height - 10, null);
} else if (ball.isStop()) {
    // 绘制游戏结束图像
    g.drawImage(gameOver.getImage(), gameOver.getX(), gameOver.getY(),
                width, height - 10, null);
} else {
    // 清除原来的图像
    g.clearRect(0, 0, width, height);
    // 绘制挡板图像
    g.drawImage(stick.getImage(), stick.getX(), stick.getY(), stick
                .getPreWidth(), stick.getImage().getHeight(null), null);
}
```

到此，我们可以运行程序查看创建挡板后的效果，具体的效果如图 4.6 所示。

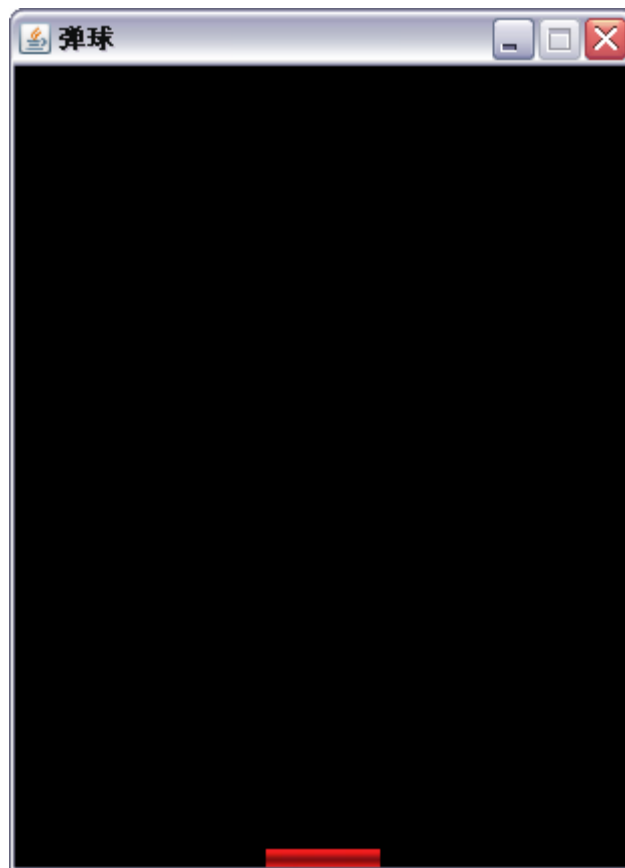


图 4.6 创建挡板

### 4.5.2 小球（Ball类）

此类提供一个以画板的宽、高、挡板高度与小球的图片路径为参数的构造器，见以下代码。

代码清单：code\ball\src\org\crazyit\ball\Ball.java

```
public Ball( int panelWidth , int panelHeight , int offset, String path )
    throws IOException {
```

```
//调用父构造器
super( panelWidth, panelHeight, path );
//设置 y 坐标
this.setY( panelHeight - super.getImage().getHeight( null ) - offset );
}
```

首先调用父类的 `BallComponent(int x, int y, String path)` 构造器，把此对象的 `x` 坐标设置到画板中间的位置，并且使用 `javax.imageio.ImageIO` 的 `read` 方法去读取磁盘中的图片文件。接下来把 `y` 坐标设置到板位上面的位置。

在这里，小球对象有两种状态，一种是小球是否开始运动，这种状态下，如果小球没有开始运动，代表准备开始游戏，反则代表游戏已经开始，没游戏没结束之前，小球就一直运动；一个是小球是否结束运动，如果小球结束运动，代表游戏已经结束，小球不能再运动，挡板也不再受玩家的控制，反则代表正在游戏中。我们在 `Ball` 中提供一个 `started` 的属性来标识这两种状态。那么当游戏开始时，就可以直接设置 `Ball` 的 `started` 属性为 `true`。

我们把小球的速度方向分为横与竖两个方向，所以这里用 `int` 类型的 `speedX` 与 `speedY` 两个属性去代表小球的横向方向与竖向方向，并增加相应的 `setter` 与 `getter` 方法。为 `Ball` 对象添加了相关的属性后，我们可以在 `BallService` 的 `draw` 方法中，将一个小球“画”到 `BallPanel` 中。具体的效果如图 4.7 所示。

```
g.drawImage(ball.getImage(), ball.getX(), ball.getY(), null);
```

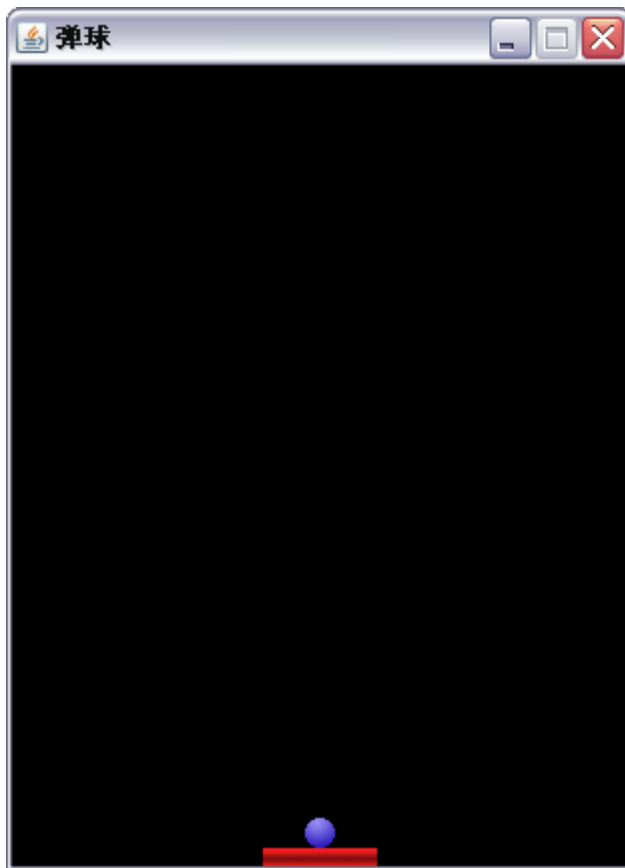


图 4.7 画小球图片

### 4.5.3 道具（Magic及其子类）

`Magic` 类是一个抽象类，它是 `BallComponet` 的子类，又是 `LongMagic` 与 `ShortMagic` 的父类，此

类只有一个抽象方法 `magicDo`，用来完成道具的功能，提供一个使用图片路径与 `x`、`y` 坐标为参数的构造器供其子类继承，见以下代码。

代码清单：code\ball\src\org\crazyit\ball\Magic.java

```
public Magic( String path, int x , int y ) throws IOException {
    super( path, x, y );
}
public abstract void magicDo( Stick stick );
```

这个构造器只调用父类 `BallComponent` 的构造器，去设置道具的表现图片与初始坐标。加粗的一行代码是用来完成道具功能的抽象方法，这里只有定义，没有实现，让其子类去实现。`Magic` 类有两个子类：`LongMagic` 与 `ShortMagic`，这两个道具的功能是使游戏中的挡板变长和变短，功能都在 `magicDo` 的实现方法中实现，首先看 `LongMagic` 类实现的 `magicDo` 方法。

代码清单：code\ball\src\org\crazyit\ball\LongMagic.java

```
public void magicDo( Stick stick ) {
    double imageWidth = stick.getImage().getWidth(null);
    //如果挡板没有变长过
    if( stick.getPreWidth() <= imageWidth ) {
        //将挡板的长度改为双倍
        stick.setPreWidth( (int)(stick.getPreWidth() * 2 ) );
    }
}
```

首先获取挡板图片的长度，再拿这个长度和挡板现在的长度比较，如果挡板的长度小于或者等于图片的长度，说明挡板的长度没有增加过，所以就调用 `Stick` 的 `setPreWidth` 方法把挡板的长度设置为又倍，下面再看 `ShortMagic` 实现的 `magicDo` 方法。

代码清单：code\ball\src\org\crazyit\ball\ShortMagic.java

```
public void magicDo( Stick stick ) {
    double imageWidth = stick.getImage().getWidth(null);
    //如果挡板没有变短过
    if( stick.getPreWidth() >= imageWidth ) {
        //将挡板的宽度改为一半
        stick.setPreWidth( (int)(stick.getPreWidth() * 0.5 ) );
    }
}
```

这里的流程和 `LongMagic` 中实现的方法相似，首先获取挡板图片的长度，如果现在的长度大于或者等于图片的长度，说明挡板的长度没有减少过，就调用 `Stick` 的 `setPreWidth` 方法把挡板的长度设置为一半。

#### 4.5.4 砖块（Brick类）

`Brick` 类是 `BallComponent` 的一个子类，用一个 `boolean` 类型的属性 `disalbe` 去标志对象是否有效果，还包含一个 `Magic` 类型的属性 `magic`，在构造器中初始化这个属性，见以下代码中。

代码清单：code\ball\src\org\crazyit\ball\Brick.java

```
public Brick(String path, int type, int x, int y ) throws IOException {
    super(path);
    if( type == Brick.MAGIC_LONG_TYPE ) {
        this.magic = new LongMagic( "img/long.gif", x, y );
    } else if( type == Brick.MAGIC_SHORT_TYPE ) {
        this.magic = new ShortMagic( "img/short.gif", x, y );
    }
    if( this.magic != null ) {
```

```

        this.magic.setX( x );
        this.magic.setY( y );
    }
}

```

在这个构造器的参数中，除了读取图片文件的 `path` 参数和对象坐标的 `x` 与 `y` 参数，还有一个 `int` 类型的参数 `type`，构造器主要是根据这个参数的值去决定此对象包含的 `Magic`，如是 `type` 等于 `Brick.MAGIC_LONG_TYPE`，`magic` 就是一个 `LongMaigc` 对象，如果 `type` 等于 `Brick.MAGIC_SHORT_TYPE`，`magic` 就是一个 `ShortMagic` 对象，如果 `magic` 不是空值，就设置 `magic` 的 `x` 与 `y` 坐标。当然，同样需要为 `magic` 与 `disalbe` 属性增加相应的 `setter` 和 `getter` 方法。

## 4.6 BallService类实现

`BallService` 被定义成一个专门处理此游戏逻辑功能的类，包含处理小球的移动、处理挡板的移动、初始化砖块与道具、判断玩家的输赢，判断游戏中的图片元素是否有碰撞，把图片绘制到画板等功能。由于 `BallService` 负责处理几乎全部的游戏逻辑，那么该类中就需要维护界面所有的组件：小球对象、挡板对象、砖块的二维数组等。`BallService` 中所有的方法都是对这些对象进行处理，修改它们的相关属性或者执行相关的行为。

### 4.6.1 创建与设置砖块

在本游戏的设计中，为了简单起见，没有加入游戏关卡的概念，没有去设置每一关的砖块与道具等东西的分布，所以，游戏开始的时候，我们会用一个方法名为 `createBrickArr` 的方法去随机产生砖块与道具，先看以下代码。

代码清单：code\ball\src\org\crazyit\ball\BallService.java

```

public Brick[][] createBrickArr( String path, int xSize, int ySize ) throws IOException {
    //创建一个 Brick[][]
    Brick[][] bricks = new Brick[xSize][ySize];
    int x = 0;
    int y = 0;
    int random = 0;
    int imageSize = 28;
    boolean isDisable = false;
    //迭代初始化数组
    for ( int i = 0 ; i < xSize ; i++ ) {
        for ( int j = 0 ; j < ySize ; j++ ) {
            //创建一个新的砖块
            random = (int)( Math.random() * 3 );
            x = i * imageSize;
            y = j * imageSize;
            //一定机率没有砖块
            isDisable = Math.random() > 0.8 ? true : false;
            if( isDisable ){
                random = 0;
            }
            Brick brick = new Brick( path, random, x, y );
            brick.setDisable(isDisable);
            //设置 x 坐标

```

```

        brick.setX( x );
        //设置 y 坐标
        brick.setY( y );
        bricks[i][j] = brick;
    }
}
return bricks;
}

```

这个方法的返回类型是 **Brick[][]**，也就是说是一个 **Brick** 类型的二维数组，**bricks[i][j]** 代表砖块在第 **i** 行第 **j** 列。有三个参数：**String** 类型的图片文件路径 **path**，还有代表返回数组大小的 **xSize** 与 **ySize**，这两个参数是 **int** 类型。首先，以 **xSize** 与 **ySize** 去创建一个 **Brick[][]** 类型的变量，接下来遍历这个数组，在遍历的过程中，每次先创建一个砖块，然后随机设置砖块对象的 **disable** 属性，**disable** 属性为 **true** 的砖块将不会被显示，创建砖块的过程中，也是随机创建砖块所包含的道具，然后再设置这个砖块的 **x** 与 **y** 坐标，最后把新创建的砖块对象赋给 **bricks[i][j]**。遍历完这个数组后，便把 **bricks** 返回。这样就完成创建游戏中所有砖块与道具的过程。

创建了砖块的二维数组后，我们就需要将这个二维数组“画”到 **BallPanel** 中，为 **BallService** 的 **draw** 加入相关的实现即可。

代码清单：code\ball\src\org\crazyit\ball\BallService.java

```

// 迭代绘制砖块图像
for (int i = 0; i < bricks.length; i++) {
    for (int j = 0; j < bricks[i].length; j++) {
        BallComponent magic = bricks[i][j].getMagic();
        // 如果这个砖块图像对象是有效的
        if (!bricks[i][j].isDisable()) {
            // 里面的数字 1 为砖块图像间的间隙
            g.drawImage(bricks[i][j].getImage(), bricks[i][j]
                .getX(), bricks[i][j].getY(), bricks[i][j]
                .getImage().getWidth(null) - 1, bricks[i][j]
                .getImage().getHeight(null) - 1, null);
        } else if (magic != null && magic.getY() < height) {
            g.drawImage(magic.getImage(), magic.getX(), magic
                .getY(), null);
        }
    }
}

```

同样地，使用嵌套循环将砖块的二维数组“画”到 **BallPanel** 中，在绘画该二维数组的时候，要判断砖块是否有效。需要注意的是，必须是游戏中时才进行绘画，当游戏结束（胜利、失败）或者小球停止运动的时候，我们并不需要绘画此二维数组。绘画砖块的具体效果如图 4.8 所示。



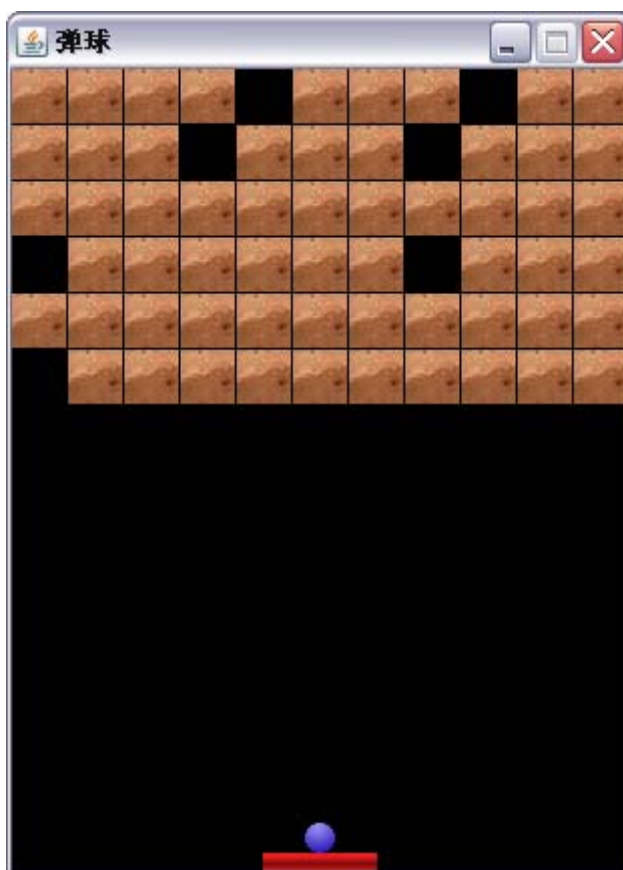


图 4.8 创建砖块

#### 4.6.2 设置挡板的位置（移动挡板）

挡板的移动主要是依靠监听玩家的键盘操作，然后做出相应的反应，去改变挡板的坐标位置，所以需要以一个 `KeyEvent` 对象做为这个方法的参数，在方法内可以通过这个对象的 `getKeyCode()` 方法去获取玩家所按下的键盘按键，先看以下代码。

代码清单：code\ball\src\org\crazyit\ball\BallService.java

```
public void setStickPos( KeyEvent ke ) {
    //把弹球的运动状态设为 true
    ball.setStarted( true );
    //如果是左方向键
    if( ke.getKeyCode() == KeyEvent.VK_LEFT ) {
        if( stick.getX() - stick.SPEED > 0 ) {
            //x 坐标向左移动
            stick.setX( stick.getX() - stick.SPEED );
        }
    }
    //如果是右方向键
    if( ke.getKeyCode() == KeyEvent.VK_RIGHT ) {
        if( stick.getX() + stick.SPEED < width
            - stick.getPreWidth() ) {
            //x 坐标向右移动
            stick.setX( stick.getX() + stick.SPEED );
        }
    }
}
```

```

        //ballFrame.getBallGame().reStart( ballFrame );
    }
}
//如果是 F2 键
if( ke.getKeyCode() == KeyEvent.VK_F2 )    {
    //初始化 ballFrame
    try {
        ballFrame.initialize();
    } catch ( IOException e ) {
        System.out.println( e.getMessage() );
    }
}
}
}

```

如果玩家按下的是左键，也就是 `ke.getKeyCode()` 等于 `KeyEvent.VK_LEFT`，就先检查挡板是否已经在游戏面板的最左边，如果不是，就把挡板的位置向左移动 `Stick.SPEED` 个位置（`SPEED` 代表挡板的移动速度），否则不做任何操作。如果玩家按下的是右键，处理方式与左键类似，只不过是方向相反。如果玩家按下的是 **F2** 键（这里定义 **F2** 键是重新开始游戏），就调用 `BallFrame` 对象的 `initialize` 方法是重新初始化界面。实现挡板的移动较为简单，只需要设置挡板对象的坐标并判断是否越界面即可。

#### 4.6.3 小球与砖块碰撞

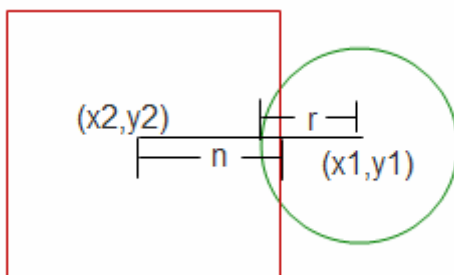


图 4.9 小球与砖块碰撞

在游戏中，如果运行的小球碰到砖块，就要把砖块消掉，所以我们需要判断小球与砖块是否有碰撞，假设小球圆心的坐标是  $(x_1, y_1)$ ，砖块中间的坐标是  $(x_2, y_2)$ ，砖块的一半边长是  $n$ ，小球的半径是  $r$ ，那么，如果  $(x_1, y_1)$  与  $(x_2, y_2)$  的距离小于  $n+r$ ，砖块与小球就处于碰撞的状态，见图 4.9 与以下代码。

代码清单：code\ball\src\org\crazyit\ball\BallService.java

```

public boolean isHitBrick( Brick brick ) {
    if ( brick.isDisable() ) {
        return false;
    }
    //ball 的圆心 x 坐标
    double ballX = ball.getX()
        + ball.getImage().getWidth(null) / 2;
    //ball 的圆心 y 坐标
    double ballY = ball.getY()
        + ball.getImage().getHeight(null) / 2;
    //brick 的中心 x 坐标
    double brickX = brick.getX()

```

```

        + brick.getImage().getWidth(null)/2;
//brick 的中心 y 坐标
double brickY = brick.getY()
        + brick.getImage().getHeight(null)/2;
//两个坐标点的距离
double distance = Math.sqrt(
        Math.pow( ballX - brickX, 2 )
        + Math.pow( ballY - brickY, 2 ));
//如果两个图形重叠, 返回 true;
if( distance < ( ball.getImage().getWidth(null)
        + brick.getImage().getWidth(null) )/ 2) {
    //使 brick 无效
    brick.setDisable( true );
    return true;
}
return false;
}

```

粗体代码部分就是以  $(x1, y1)$ ,  $(x2, y2)$  两个点的距离与  $n$ 、 $r$  的和比较, 如果这个距离小于和, 就调用 **Brick** 对象的 **setDisable** 方法把 **Brick** 对象设置为无效, 并返回 **true**。我们将砖块的二维数组“画”到 **BallPanel** 中的时候 (遍历二维数组), 得到每一个砖块对象, 都需判断该对象的 **disable** 属性, 如果该属性为 **true**, 则表示这块砖块仍然处在原来的位置, 如果该属性为 **false**, 则表示这块砖块已经被小球碰撞, 并出“跌落”了相应的道具, 在 **draw** 的时候, 就需要将道具的图片画到界面中 (**BallPanel**), 小球碰撞砖块的效果如图 4.10 所示。

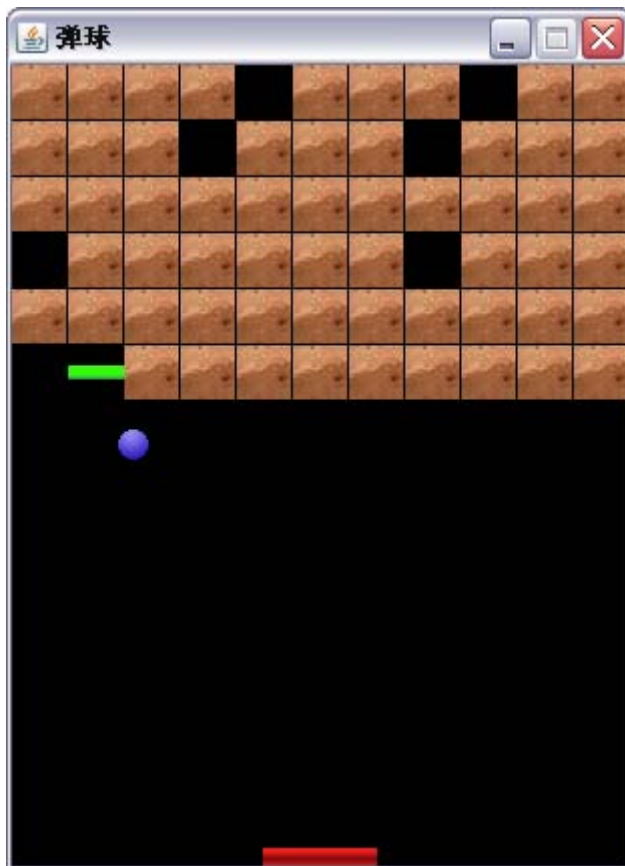


图 4.10 小球与砖块碰撞

如图 4.10 所示，当小球与砖块发生碰撞的时候，砖块就会变成道具，并且该道具会进行下落。道具的移动、道具与挡板的碰撞我们将在下面的章节中描述。

#### 4.6.4 小球、道具与挡板碰撞

我们需要实现 **BallService** 的 **isHitStick** 方法，该方法判断小球、道具与挡板是否发生了碰撞，只要它们发生了碰撞，该方法就需要返回 **true**。**isHitStick** 方法只要判断是否发生了碰撞，至于发生碰撞后所需要处理的事情，并不由该方法进行处理。在这里，由于挡板是长方形的，而且挡板的 **y** 坐标是不变的，所以可以不使用上节判断小球与砖块碰撞的方法。假设挡板的坐标是指这个长方形的左上角，用 (**x1**, **y1**) 表示，挡板的长度为 **n**，那么，只要小球或者道具的 **x** 坐标处于 **x1** 与 **x1+n** 之间（也就是处于挡板的范围内），**y** 坐标大于 **y1**，那么就可以判断它们在碰撞，见以下代码。

代码清单：code\ball\src\org\crazyit\ball\BallService.java

```
public boolean isHitStick( BallComponent bc ) {
    //获取图片对象
    Image templImage = bc.getImage();
    //如果与挡板有碰撞
    if( bc.getX() + templImage.getWidth(null) > stick.getX()
        && bc.getX() < stick.getX() + stick.getPreWidth()
        && bc.getY() + templImage.getHeight(null) > stick.getY() ) {
        return true;
    }
    return false;
}
```

这个方法中的参数 **bc** 代表的是小球或者道具，加粗代码部分是判断它们是否有碰撞，**bc.getX() + templImage.getWidth(null) > stick.getX()** 是确认 **bc** 的 **x** 坐标是不是处于挡板的范围内，**bc.getY() + templImage.getHeight(null) > stick.getY()** 是确认 **bc** 的 **y** 坐标是否大于挡板的 **y** 坐标。

#### 4.6.5 道具的移动

当小球与砖块发生碰撞后，砖块将会变成道具（如图 4.10 所示）。前面的章节中讲到这个游戏有两个道具，**LongMagic** 与 **ShortMagic**，作用分别是使挡板就长或者变短，而道具是保存在 **Brick** 对象中，所以我们需要遍历 **bricks** 数组中的所有 **Brick** 对象，如果 **Brick** 对象的状态是 **disable** 为 **true**（也就是说砖块被小球消掉），而且这个 **Brick** 对象中有 **Magic** 对象不为 **null**，并且 **Magic** 对象的 **y** 坐标小于画板的高度 **height**（这里意思是说这个道具还在画板的范围之内），那么，便以 **Magic** 对象的速度每次增加 **magic.getSpeed()** 个 **y** 坐标值，达到道具向下移动的效果，见如下代码。

代码清单：code\ball\src\org\crazyit\ball\BallService.java

```
public void setMagicPos() {
    for ( int i = 0 ; i < bricks.length ; i++ ) {
        for ( int j = 0 ; j < bricks[i].length ; j++ ) {
            //获取 magic
            Magic magic = bricks[i][j].getMagic();
            if( magic != null ) {
                //如果这个 brick 的状态是无效的
                if( bricks[i][j].isDisable() && magic.getY() < height ) {
                    //设置 magic 的 y 坐标向下增加
                    magic.setY( magic.getY() + magic.getSpeed() );
                    //设置挡板的宽度
                }
            }
        }
    }
}
```

```

        setStickWidth( magic );
    }
}
}
}
}

```

以上的代码实现了 `setMagicPos` 方法，该方法每执行一次，都会改变道具的位置，因此，我们可以在 `BallService` 的 `run` 方法调用 `setMagicPos` 方法（`run` 方法每 100 毫秒执行一次），如果砖块被消除的话，界面中就会出现下落的道具，具体的效果如图 4.11 所示。

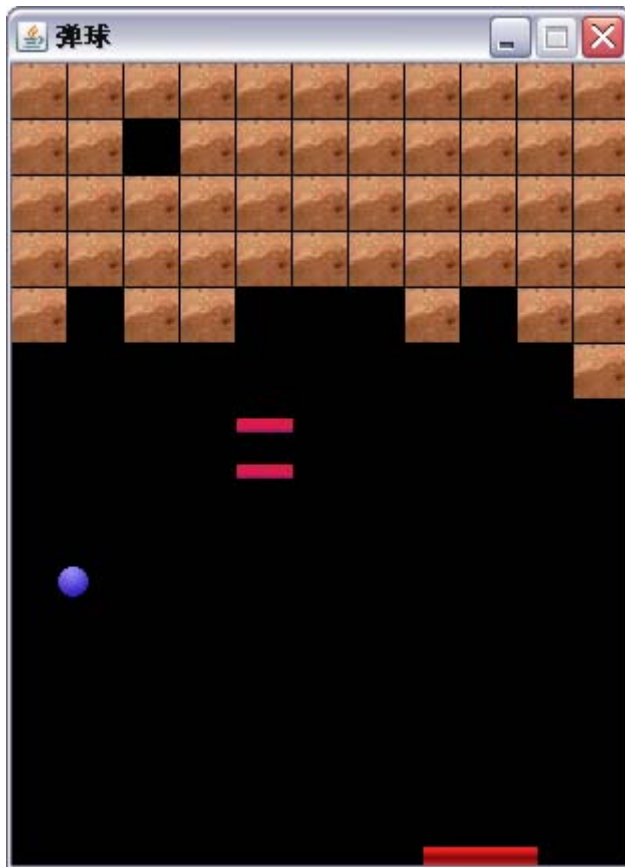


图 4.11 道具的下落

#### 4.6.6 改变挡板的长度（道具的作用）

在 4.6.4 中实现了判断小球与挡板、砖块是否发生碰撞的方法，因此在这里改变挡板长度，实现起来将会十分简单，只要判断道具与挡板是否有碰撞（调用 `isHitStick` 方法），如果挡板与“掉下来”的道具发生碰撞，便调用 `Magic` 对象的 `magicDo` 方法，`magicDo` 方法会将挡板的长度，见以下代码。

代码清单：code\ball\src\org\crazyit\ball\BallService.java

```

public void setStickWidth( Magic magic ) {
    if( isHitStick( magic ) ) {
        //道具的作用
        magic.magicDo( stick );
    }
}

```

在本章中只涉及两个道具：缩短挡板和加长挡板，并且我们在 **Magic** 的两个子类中已经对其提供了相应的实现，因此我们在以上的代码中可以直接调用 **Magic** 的 **magicDo** 方法，这样就可以将当前的挡板加长或者缩短。**Magic** 的两个子类已经在 4.5.3 中实现，接下来我们可以运行游戏查看效果，游戏的具体效果如图 4.12 所示。

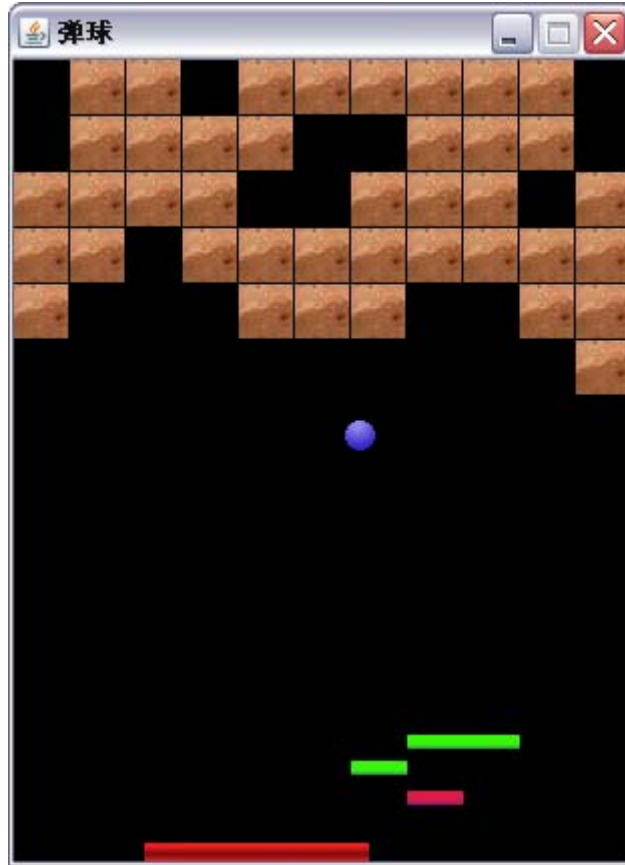


图 4.12 道具的效果

如图 4.12 所示，当游戏中的挡板接收到绿色的道具时（加长挡板），挡板的宽度就发生了改变，这是由于我们在 **Magic** 的子类（**LongMagic**）中设置了挡板的宽度。

#### 4.6.7 判断是否已经通关

在本游戏中，是否通关的标准是，小球是否已经将所有砖块清除。因此我们实现该功能的时候，就需要在 **BallService** 中对砖块的二维数组进行遍历，如果该数组中所有的砖块对象的 **disable** 属性都为 **true** 的话，就意味着所有的砖块都已经被“击落”，这一关游戏通过；如果数组中的某一个砖块的 **disable** 属性为 **false** 的话，游戏需要继续进行。

代码清单：code\ball\src\org\crazyit\ball\BallService.java

```
public boolean isWon() {  
    //如果消了全部砖块，则为赢  
    for (int i = 0 ; i < bricks.length ; i++) {  
        for (int j = 0 ; j < bricks[i].length ; j++) {  
            if(!bricks[i][j].isDisable()) {  
                return false;  
            }  
        }  
    }  
}
```

```
}  
    return true;  
}
```

实现了 `isWon` 方法后，我们可以启动游戏进行测试，由于我们在前面章节实现 `draw` 方法的时候，就需要调用 `isWon` 方法来判断游戏是否胜利，如果胜利的话，就需要将游戏胜利的图片“画”到 `BallPanel` 中，当游戏胜利时，具体的效果如图 4.13 所示。



图 4.13 游戏胜利

到此，我们的桌面弹球游戏就已经全部实现了，可以运行游戏进行测试。我们这个桌面弹球并没有任何复杂的功能，只是在小球运动的过程中将砖块消除并转化成相应的道具，最后判断游戏是否已经胜利，我们可以在此基础上开发出更多有趣的功能，例如加入分数的计算、加入多种道具等。

## 4.7 功能改进设计

从前面的几小节可以看到，本游戏并没有加入关卡、计分等概念，而且道具的种类比较少，整个游戏玩起来会比较枯燥，其实我们不必做太多的工作，就能把这些概念加到游戏中去。如果我们需要加入更多的游戏道具，可以继承 `Magic` 对象并实现 `magicDo` 方法。如果需要更改砖块的排列，可以编写程序动态创建砖块的二维数组。

### 4.7.1 关卡

我们可以设计每一关砖块的不同排列方式，与及里面所包括的道具，可以怎么去设置这个东西？好，

我们可以使用最简单的一个 **txt** 文本去设置每关的砖块与道具，例如使用一个 “\*” 号去代表砖块，然后在 “\*” 号旁边加上一个小括号去加入道具的类型，见以下文档：

```
* (0) * (0) * (0) * (0) * (0) * (0)
* (0) * (0) * (1) * (0) * (0) * (4)
* (0) * (0) * (0) * (0) * (0) * (0)
* (0) * (0) * (0) * (0) * (0) * (0)
* (0) * (0) * (0) * (2) * (0) * (0)
* (3) * (0) * (0) * (0) * (0) * (0)
* (0) * (0) * (0) * (0) * (0) * (0)
* (0) * (0) * (0) * (0) * (0) * (0)
```

然后我们就可以通过 **IO** 操作去读取与分析这个文件，分析后便设置到 **Brick** 类型的数组里面。

### 4.7.2 计分

可以简单实现一个即时计分的功能也比较简单，可以给在 **BallGame** 类中增加一个静态的 **int** 属性去保存分数，然后再砖块与各个道具类中增加一个分数的属性，如果有碰撞的时候，便把这些分数加到 **BallGame** 的计分器中。

### 4.7.3 道具

由于我们的道具使用了一个 **Magic** 抽象类作为接口，增加新的道具也比较简单，例如我现在要增加一个一次能把所有砖块清掉的道具，那么，我可以新建一个叫 **KillAllMagic** 的类，继承 **Magic** 类，并实现里面的 **doMagic** 方法，见以下代码：

```
public void magicDo( Stick stick ) {
    for ( int i = 0 ; i < bricks.length ; i++ ) {
        for ( int j = 0 ; j < bricks[i].length ; j++ ) {
            bricks[i][j].setDisable(true);
        }
    }
}
```

把所有的砖块都设置为无效果，就清除掉所有的砖块了。

## 4.8 本章小结

本章主要是通过一个弹球游戏的基本实现，向读者讲解 **Java** 的画图方法，主要是使用 **Graphics** 对象的 **drawImage** 方法去画图。在开发桌面弹球的过程中，我们将界面中的砖块抽象成一个二维数组，将游戏中的相关组件（小球、挡板）都抽象成为一个对象，并为 **JFrame** 提供了键盘监听器，当监听器接收到按键信息后，就会调用相关的方法去操作游戏中的各个对象，并将这些对象画到界面中。本章主要详细描述了键盘事件监听器、在 **Swing** 组件中画图等相关知识点。



## 第5章 单机俄罗斯方块

### 5.1 俄罗斯方块简介

俄罗斯方块是我们最常见的游戏之一，该游戏出现过在掌上游戏机、家用游戏机、手机游戏和电脑游戏中，因此俄罗斯方块也是一个十分经典的游戏。我们可以在网上下载到各式各样的俄罗斯方块游戏，也可以在各个游戏大厅中见到网络对战形式的俄罗斯方块。一般的俄罗斯方块规则比较简单，游戏中随机出现一些方块，再对这些方块进行变换，下降到游戏界面中的不同位置，如果某一行中都填充了方块，那么该行就消除。当然，还有更复杂的俄罗斯方块，例如方块穿墙，方块消除列等。本章中我们实现一个简单的单机俄罗斯方块，让大家可以了解俄罗斯的实现原理。

### 5.2 建立界面

俄罗斯方块的界面包括两个部分，一个是方块堆砌的界面，另一部分是工具界面，工具界面包括一些游戏计分和游戏控制的功能。在本章中，我们只需要提供简单的暂停和开始功能，如果需要做到更为复杂的俄罗斯方块，还要提供工具栏，让用户进行各种操作。

#### 5.2 方块堆砌界面

在本章中，方块堆砌界面使用一个 `GamePanel` 来实现，整个游戏界面使用一个 `MainFrame` 的类来实现，`MainFrame` 继承于 `JFrame`，`GamePanel` 继承于 `JPanel`。`GamePanel` 需要重写父类的 `paint` 方法，将背景的图片画到 `GamePanel` 中。

代码清单：code\tetris\src\org\crazyit\tetris\ui\GamePanel.java

```
public void paint(Graphics g) {  
    g.drawImage(this.background, 0, 0, this.getWidth(),  
        this.getHeight(), null);  
}
```

我们可以将 `GamePanel` 看作是一个二维数组，俄罗斯方块中的每一个小方块看作是这个二维数组中的每一个元素，然后在 `paint` 方法中将这些小方块画到 `GamePanel` 中，这些将在下面的章节中描述。当一个大方块进行一次下降的时候，`GamePanel` 就根据这个二维数组进行一次重绘（调用 `repaint` 方法）。

#### 5.3 游戏界面

整个游戏界面包括一个 `GamePanel` 和另外的一个 `JPanel`，该 `JPanel` 代表工具界面，包括分数的计算、当前游戏级别的显示、游戏的操作和显示下一个大方块。

代码清单：code\tetris\src\org\crazyit\tetris\ui\MainFrame.java

```
public MainFrame() {  
    BoxLayout toolPanelLayout = new BoxLayout(this.toolPanel, BoxLayout.Y_AXIS);  
    this.toolPanel.setLayout(toolPanelLayout);  
    this.toolPanel.setBorder(new EtchedBorder());  
}
```

```
this.toolPanel.setBackground(Color.gray);  
//分数  
this.scoreTextBox.add(this.scoreTextLabel);  
this.scoreLabel.setText(String.valueOf(this.score));  
this.scoreBox.add(this.scoreLabel);  
//级别  
this.levelTextBox.add(this.levelTextLabel);  
this.levelLabel.setText(String.valueOf(this.currentLevel));  
this.levelBox.add(this.levelLabel);  
//继续按钮  
this.resumeLabel.setIcon(RESUME_ICON);  
this.resumeLabel.setPreferredSize(new Dimension(3, 25));  
this.resumeBox.add(this.resumeLabel);  
//暂停按钮  
this.pauseLabel.setIcon(PAUSE_ICON);  
this.pauseLabel.setPreferredSize(new Dimension(3, 25));  
this.pauseBox.add(this.pauseLabel);  
//开始  
this.startLabel.setIcon(START_ICON);  
this.startLabel.setPreferredSize(new Dimension(3, 25));  
this.startBox.add(this.startLabel);  
//下一个  
this.nextTextBox.add(this.nextTextLabel);  
//省略其他布局代码  
}
```

最后将 **GamePanel** 和工具界面的 **JPanel** 分别放置到 **MainFrame** 中不同位置:

```
this.add(this.gamePanel, BorderLayout.CENTER);  
this.add(this.toolPanel, BorderLayout.EAST);
```

以上代码中的 **this** 表示 **MainFrame**。最后界面的效果如图 5.1 所示。

*www.docin.com*



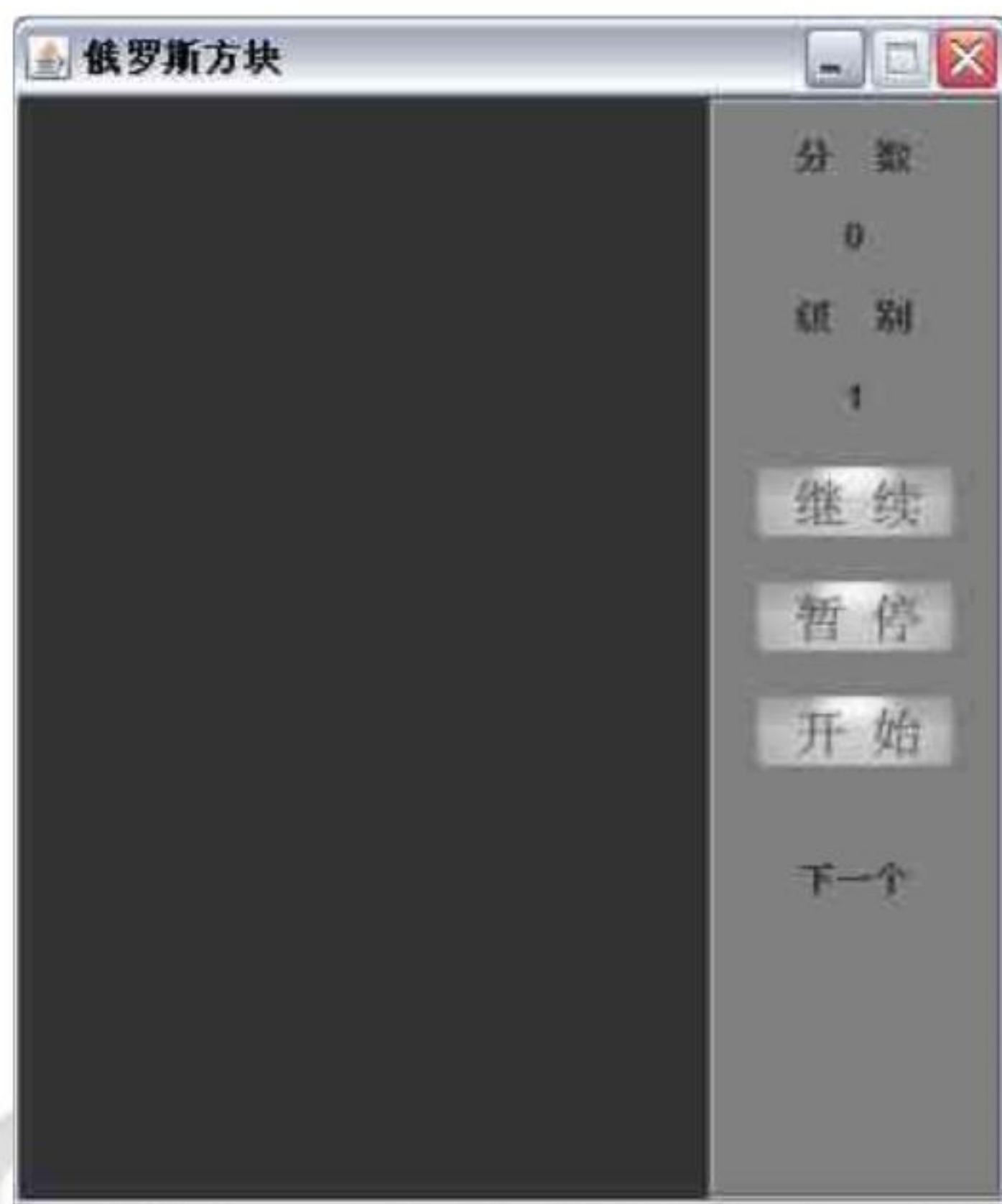


图 5.1 游戏主界面

其中继续、暂停和开始的三个按钮我们使用 `JLabel` 实现，当鼠标经过这三个按钮的时候，就改变这三个按钮的图片，因此需要加入鼠标监听器。

代码清单：`code\tetris\src\org\crazyit\tetris\ui\MainFrame.java`

```
this.resumeLabel.addMouseListener(new MouseAdapter() {  
    public void mouseEntered(MouseEvent e) {  
        resumeLabel.setIcon(RESUME_ON_ICON);  
    }  
    public void mouseExited(MouseEvent e) {  
        resumeLabel.setIcon(RESUME_ICON);  
    }  
    public void mouseClicked(MouseEvent e) {  
        //继续方法  
    }  
});
```

当鼠标经过“继续”的 `JLabel` 时，就会触发以上代码中的 `mouseEntered` 方法，将图片更改，如果鼠标离开 `JLabel` 的时候，就会触发 `mouseExited` 方法，使用 `JLabel` 显示的图片变回原来的图片。

### 5.3 创建游戏对象

在开始实现游戏前，我们可以先设计游戏中的各个对象，例如一个大方块我们可以将其抽象成一个对象，而这个大方块是由各个小方块组成的，因此又可以将各个小方块抽象成一个对象。那么在 `GamePanel` 中就可以将正在下降的大方块与已经下降的小方块通过 `paint` 方法展现到界面中。



### 5.3.1 设计小方块对象

新建一个 **Square** 对象来表示一个小方块，那么该小方块对象就需要包括三个属性，分别是该方块的图片、开始横坐标和开始纵坐标。

代码清单：code\tetris\src\org\crazyit\tetris\object\Square.java

```
public class Square {  
    //方块的图片  
    private Image image;  
    //开始横坐标  
    private int beginX;  
    //开始纵坐标  
    private int beginY;  
    //省略 setter 和 getter 方法  
}
```

在 **Square** 对象中，我们并不需要提供大方块对象的属性，也就是说小方块并不需要知道自己是属于哪个大方块的，当大方块在下降的过程中，小方块就是属于大方块，而一旦大方块下降完成后，这种关系就不再存在，因此我们可以将小方块与大方块之间的关系放置到大方块的对象中。

### 5.3.2 设计大方块对象

大方块对象包括多个小方块，而这些小方块的不同位置都使得大方块产生不同的变化，每一个大方块都有多种变化。新建一个 **Piece** 对象来表示一个大方块，**Piece** 包括如下属性。

代码清单：code\tetris\src\org\crazyit\tetris\object\Piece.java

```
public class Piece {  
    //该大方块所包含的小方块  
    private List<Square> squares;  
    //该大方块的变化  
    protected List<List<Square>> changes = new ArrayList<List<Square>>();  
    //当前变化的索引（在 changes 集合中的索引）  
    protected int currentIndex;  
    //每个小块的边长  
    public final static int SQUARE_BORDER = 16;  
}
```

在 **Piece** 对象中，需要特别注意的是 **currentIndex** 属性，该属性表示当前大方块的变化，而大方块的变化用一个 **changes** 的集合来保存，这个集合中存放各种变化，每一种变化又包含多个小方块对象（**Square**）。如果需要创建一个大方块，就需要新建一个类去继承 **Piece** 类，在构造器中设定各个小方块位置，例如我们需要创建一个图 5.2 的正方形大方块。



图 5.2 正方形大方块

编写一个 **Piece0** 的类，并在构造器中设定各个小方块的位置。

代码清单：code\tetris\src\org\crazyit\tetris\piece\Piece0.java

```
public class Piece0 extends Piece {  
    public Piece0(Image image) {  
        //创建各个小方块，一个集合为一种变化  
        List<Square> squares = new ArrayList<Square>();  
        squares.add(new Square(image, 0, 0));  
    }  
}
```



```

squares.add(new Square(image, 0, image.getHeight(null)));
squares.add(new Square(image, image.getWidth(null), 0));
squares.add(new Square(image, image.getWidth(null), image.getHeight(null)));
//加入到变化中
super.changes.add(squares);
super.setSquares(squares);
}
}

```

由于正方形的大方块是没有变化的，因此 **Piece** 对象中 **changes** 集合只有一个元素，表示该大方块只会有一种变化，最后调用 **Piece** 的 **setSquares** 方法，将唯一的一种变化设置到 **Piece** 对象中，那么这种变化就是该大方块对象的默认变化。默认变化，就是大方块第一次出现时的变化。这里需要注意的是，在构造 **Square**（小方块对象）的时候，需要提供该对象的开始横坐标和开始纵坐标，由于在构造该小方块对象的时候，这个小方块并不需要任何一个界面组件，因此可以将左上角的一个小方块的坐标看成（0，0），而其他坐标则相应加上小方块的边长（宽）。接下来，再新建一个有多种变化的大方块，如图 5.3 所示。



图 5.3 多种变化的大方块

如图 5.3 中的大方块，这种方块存在于四种变化，也就是说 **Piece** 对象中的 **changes** 集合需要保存有四个元素（四个集合）。新建一个 **Piece1** 对象，与 **Piece0** 对象与样，提供构造器。

代码清单：code\tetris\src\org\crazyit\tetris\piece\Piece1.java

```

public Piece1(Image image) {
    //第一种变化
    List<Square> squares0 = new ArrayList<Square>();
    squares0.add(new Square(image, image.getWidth(null), image.getHeight(null)));
    squares0.add(new Square(image, 0, image.getHeight(null)*2));
    squares0.add(new Square(image, image.getWidth(null), image.getHeight(null)*2));
    squares0.add(new Square(image, image.getWidth(null)*2, image.getHeight(null)*2));
    //第二种变化
    List<Square> squares1 = new ArrayList<Square>();
    squares1.add(new Square(image, 0, 0));
    squares1.add(new Square(image, 0, image.getHeight(null)));
    squares1.add(new Square(image, image.getWidth(null), image.getHeight(null)));
    squares1.add(new Square(image, 0, image.getHeight(null)*2));
    //第三种变化
    List<Square> squares2 = new ArrayList<Square>();
    squares2.add(new Square(image, 0, 0));
    squares2.add(new Square(image, image.getWidth(null), 0));
    squares2.add(new Square(image, image.getWidth(null)*2, 0));
    squares2.add(new Square(image, image.getWidth(null), image.getHeight(null)));
    //第四种变化
    List<Square> squares3 = new ArrayList<Square>();
    squares3.add(new Square(image, image.getWidth(null), image.getHeight(null)));
    squares3.add(new Square(image, image.getWidth(null)*2, 0));
    squares3.add(new Square(image, image.getWidth(null)*2, image.getHeight(null)));
    squares3.add(new Square(image, image.getWidth(null)*2, image.getHeight(null)*2));
    super.changes.add(squares0);
    super.changes.add(squares1);
}

```



```

super.changes.add(squares2);
super.changes.add(squares3);
//随机获得变化
super.setSquares(getDefault());
}

```

该大方块存在有四种变化，就为其提供四个变化集合，还需要定义各个变化的小方块位置，最后加入到 **Piece** 对象的 **changes** 集合中。以上的黑体代码中，调用了 **Piece** 对象的 **getDefault** 方法到随机取得默认的变化，那么在创建 **Piece1** 对象的时候，就可随机从四种变化中得到任意一种变化作为默认变化。

代码清单：code\tetris\src\org\crazyit\tetris\object\Piece.java

```

public List<Square> getDefault() {
    //从 changes 集合中随机得到其中一种变化
    int defaultChange = random.nextInt(changes.size());
    //设置当前变化的索引
    this.currentIndex = defaultChange;
    return changes.get(defaultChange);
}

```

通过 **getDefault** 方法，就可以在大方块构造的时候，随机获得变化。**Piece1** 的构造器中创建了四种变化，分别以 **squares0**、**squares1**、**squares2** 和 **squares3** 来代表。图 5.3 代表的是 **squares0** 定义的变化，图 5.4 代表的是 **squares1** 定义的变化，图 5.5 代表的是 **squares2** 定义的变化，图 5.6 代表的是 **squares3** 定义的变化。



图 5.4 squares1 的变化



图 5.5 squares2 的变化



图 5.6 squares3 的变化

编写了 **getDefault** 方法后，我们还需要明白，如果该 **Piece** 对象进行变化的时候，是从 **changes** 集合中得到下一个变化，再调用 **setSquares** 方法来设置 **Piece** 的 **squares** 属性。由于我们提供了当前的变化索引 (**currentIndex**)，因此就很容易得到 **changes** 的下一个。按照以上创建 **Piece0** 和 **Piece1** 的方法，来创建各种你需要的 **Piece**，那么游戏中就根据这些 **Piece** 的子类来创建大方块。通过观察我们可以发现，每一个 **Piece** 的子类都是通过创建不同的变化来创建的，如果需要做得更加灵活，我们可以将这些变化通过数据库或者配置文件来进行保存，只需要编写程序去得到这些变化的信息就可以创建各种不同的大方块。本章中如果需要创建一个大方块，就需要重新编写一个 **Piece** 的子类去实现。

## 5.4 创建与显示大方块

在 5.3 中，我们已经建立了各个大方块对象，在本节，我们编写程序去创建这些大方块对象。当游戏开始时，就需要创建一个当前的大方块，还需要创建下一个大方块。为了让每次出现的大方块都拥有不同的颜色和形状，在 5.3.2 创建 **Piece** 对象时，就已经提供了一个 **getDefault** 方法，用来随机获得某



一种大方块的默认变化。新建一个 `PieceCreator` 的接口，专门用于创建大方块，并为该接口提供一个 `PieceCreatorImpl` 的实现类。

### 5.4.1 随机读取小方块图片

为了不用每次都去读取一个 `Image` 对象，我们可以在 `PieceCreatorImpl` 中提供一个 `Map` 来保存读取过的 `Image` 对象。

代码清单：`code\tetris\src\org\crazyit\tetris\object\impl\PieceCreatorImpl.java`

```
private Map<Integer, Image> images = new HashMap<Integer, Image>();
```

而存在于文件系统的各个小方块图片，我们统一使用规定好的格式作为图片名称，方块图片的名称如图 5.7 所示。

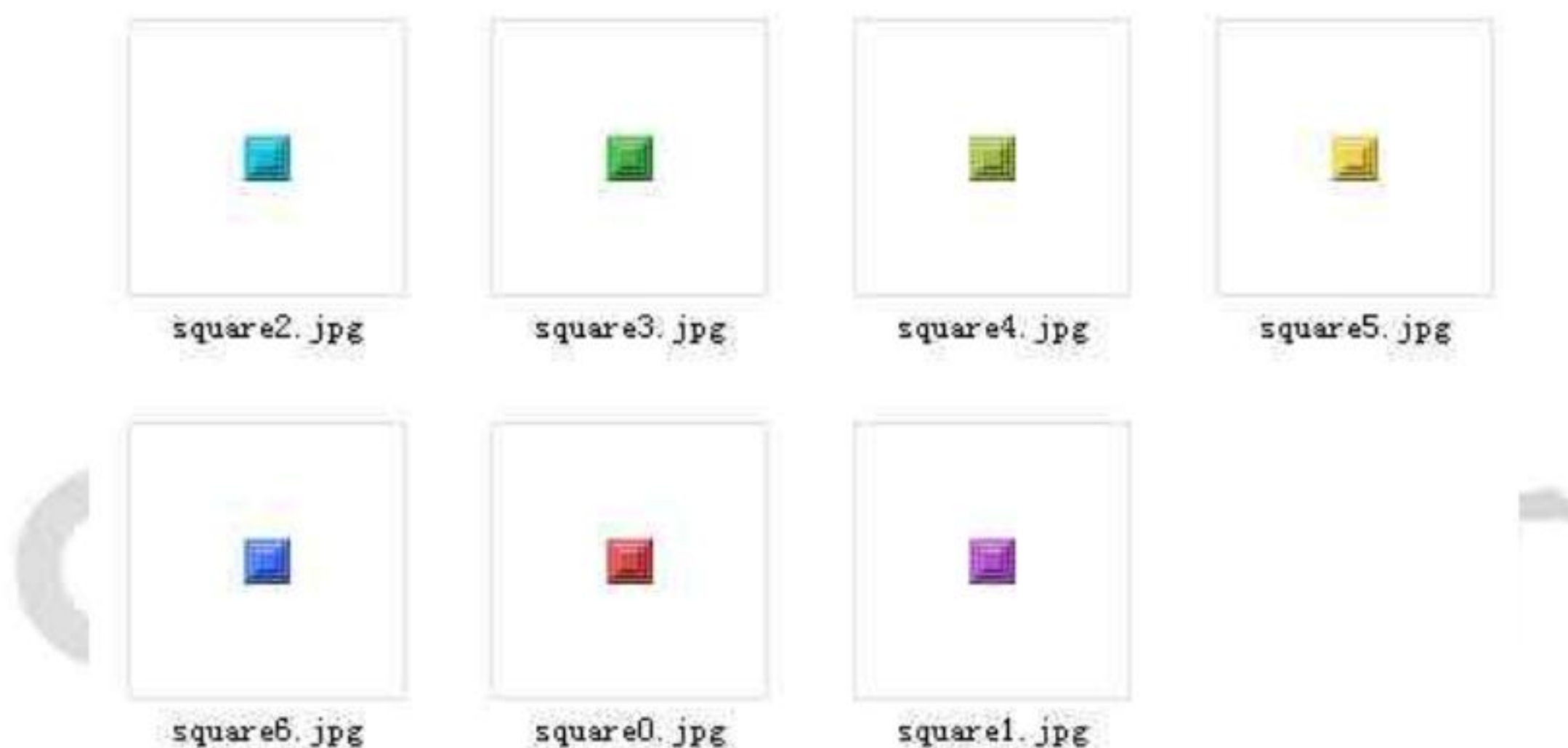


图 5.7 方块图片的名称

由于每个方块图片的名称都是“square”加上具体某个数字，所以我们只需要随机获得后面的数字，就可以实现每次都得到不同颜色的小方块图片了。

代码清单：`code\tetris\src\org\crazyit\tetris\object\impl\PieceCreatorImpl.java`

```
//从 map 中得到图片对象，如果 map 中没有存在图片对象，则创建
private Image getImage(int key) {
    if (this.images.get(key) == null) {
        Image s = ImageUtil.getImage("images/square" + key + ".jpg");
        this.images.put(key, s);
    }
    return this.images.get(key);
}
```

以上的 `getImage` 方法，从 `Map` 中找图片，如果没有，则读取。需要取哪种颜色的图片，完全由参数决定，因此我们可以让 `getImage` 的参数从随机数中取得。

```
//随机得到一张方块图片
```

```
Image image = getImage(random.nextInt(COLOR_SIZE));
```

以上代码可以随机得到一张小方块的图片，`COLOR_SIZE` 是小方块图片的总数量。



### 5.4.2 创建大方块对象

在 5.3.2 中，我们创建了各个 **Piece** 的子类，每个子类都提供一个构造器，这些构造器中都带有一个 **Image** 的构造参数，这个 **Image** 对象代表一个小方块的图片，只需要得到一个小方块的图片，就可以直接创建一个大方块对象。

在 **PieceCreator** 接口中提供一个 **createPiece** 方法，并在 **PieceCreatorImpl** 中提供实现，该方法用于创建 **Piece** 对象。

代码清单：code\tetris\src\org\crazyit\tetris\object\impl\PieceCreatorImpl.java

```
public Piece createPiece(int x, int y) {
    //随机得到一张方块图片
    Image image = getImage(random.nextInt(COLOR_SIZE));
    Piece piece = initPiece(image);
    return piece;
}
//初始化一个 Piece 对象
private Piece initPiece(Image image) {
    Piece piece = null;
    int pieceType = random.nextInt(SQUARE_SIZE);
    //初始化 Piece，随机创建各个大方块
    if (pieceType == 0) {
        piece = new Piece0(image);
    } else if (pieceType == 1) {
        piece = new Piece1(image);
    }
    //加入其他 Piece 对象
    return piece;
}
```

以上代码实现了 **PieceCreator** 的 **createPiece** 方法，调用了 **initPiece** 方法去随机创建一个具体的 **Piece** 对象。如果随机数为 0，则创建一个正方形的大方块，如果是 1，则创建一个图 5.3 中的大方块。如果创建有其他大方块，就需要再加入判断。**createPiece** 方法中的两个参数我们并没有使用，这两个参数表示在哪个位置创建一个 **Piece** 对象，这由显示 **Piece** 的 **MainFrame** 来决定。例如，需要在界面中的 (10, 10) 坐标来显示 **Piece**，那么就需要将整个 **Piece** 对象向 X 轴与 Y 轴方向各移 10。在设计 **Piece** 对象的时候，**Piece** 就不会去保存坐标，由小方块对象来保存坐标 (**Square**)，如果需要改变 **Piece** 的坐标，就意味着要改变 **Piece** 下面所有 **Square** 对象的坐标。

代码清单：code\tetris\src\org\crazyit\tetris\object\Piece.java

```
//让 Piece 对象中的所有 Square 对象的 x 坐标都加上参数 x
public void setSquaresXLocation(int x) {
    for (int i = 0; i < this.changes.size(); i++) {
        List<Square> change = this.changes.get(i);
        for (int j = 0; j < change.size(); j++) {
            Square s = change.get(j);
            s.setBeginX(s.getBeginX() + x);
        }
    }
}
```

以上的 **setSquaresXLocation** 方法，让 **Piece** 中的所有 **Square** 对象的开始横坐标加上 x (参数)，即在界面中右移。注意，需要处理所有变化中的所有小方块。除了加上横坐标，还需要加上纵坐标。

代码清单：code\tetris\src\org\crazyit\tetris\object\Piece.java

```
//让 Piece 对象中的所有 Square 对象的 y 坐标都加上参数 y
```



```

public void setSquaresYLocation(int y) {
    for (int i = 0; i < this.changes.size(); i++) {
        List<Square> change = this.changes.get(i);
        for (int j = 0; j < change.size(); j++) {
            Square s = change.get(j);
            s.setBeginY(s.getBeginY() + y);
        }
    }
}

```

通过以上的设置 **Piece** 横坐标和纵坐标的方法，就可以设置整个大方块在界面中的位置。那么在创建大方块的时候，就可以调用这两个方法来设置大方块的初始位置。

代码清单：code\tetris\src\org\crazyit\tetris\object\impl\PieceCreatorImpl.java

```

public Piece createPiece(int x, int y) {
    //随机得到一张方块图片
    Image image = getImage(random.nextInt(COLOR_SIZE));
    Piece piece = initPiece(image);
    piece.setSquaresXLocation(x);
    piece.setSquaresYLocation(y);
    return piece;
}

```

以上代码的黑体部分，就设置了一个大方块创建时的初始位置。

### 5.4.3 显示当前方块

在实现了 **PieceCreator** 创建 **Piece** 的方法后，就可以在 **MainFrame** 中提供一个 **Piece** 对象来代表当前正在下降的大方块，再提供一个 **Piece** 对象来代表下一个大方块对象。

代码清单：code\tetris\src\org\crazyit\tetris\ui\MainFrame.java

```

//当前正在运动的对象
private Piece currentPiece;
//下一个大方块对象
private Piece nextPiece;

```

并需要为当前正在运动的大方块对象提供一个 **get** 方法，为 **GamePanel** 提供一个构造器，在构造 **GamePanel** 的时候，将 **MainFrame** 作为构造参数传入。

代码清单：code\tetris\src\org\crazyit\tetris\ui\GamePanel.java

```

MainFrame mainFrame;
public GamePanel(MainFrame mainFrame) {
    this.mainFrame = mainFrame;
}

```

在 **GamePanel** 中我们可以根据 **MainFrame** 对象得到当前界面中正在运动的 **Piece** 对象，接下来继续实现 **paint** 方法，将当前运动的 **Piece** 对象也画到 **GamePanel** 中。

为 **ImageUtil** 加入一个画 **Piece** 的工具方法，该方法可以进行重用。

代码清单：code\tetris\src\org\crazyit\tetris\util\ImageUtil.java

```

//在界面上画一个 Piece 对象
public static void paintPiece(Graphics g, Piece piece) {
    if (piece == null) return;
    for (int i = 0; i < piece.getSquares().size(); i++) {
        Square s = piece.getSquares().get(i);
        //得到各个小方块后，将这些小方块画到界面中
        g.drawImage(s.getImage(), s.getBeginX(), s.getBeginY(), null);
    }
}

```

```
    }  
}
```

代码清单: code\tetris\src\org\crazyit\tetris\ui\GamePanel.java

```
public void paint(Graphics g) {  
    //画背景  
    g.drawImage(this.background, 0, 0, this.getWidth(), this.getHeight(), null);  
    //画当前运动的大方块  
    Piece currentPiece = this.mainFrame.getCurrentPiece();  
    ImageUtil.paintPiece(g, currentPiece);  
}
```

实现 **MainFrame** 的开始游戏的方法, 当开始游戏时, 就创建当前块和下一块。

代码清单: code\tetris\src\org\crazyit\tetris\ui\MainFrame.java

```
//下一个 Piece 的位置  
private final static int NEXT_X = 270;  
private final static int NEXT_Y = 320;  
//当前 Piece 的开始 X 座标  
private final static int BEGIN_X = Piece.SQUARE_BORDER * 6;  
//当前 Piece 的开始 Y 座标  
private final static int BEGIN_Y = 0;  
//创建下一个  
private void createNextPiece() {  
    this.nextPiece = this.creator.createPiece(NEXT_X, NEXT_Y);  
    this.repaint();  
}  
//开始游戏  
public void start() {  
    //创建下一个大方块  
    createNextPiece();  
    //创建当前运动的大方块  
    this.currentPiece = creator.createPiece(BEGIN_X, BEGIN_Y);  
}
```

当点击开始“按钮”时, 就调用 **start** 方法开始游戏, 创建当前大方块与下一个大方块, 并显示到界面的相应位置, 具体的效果如图 5.8 所示。





图 5.8 游戏开始

## 5.5 处理方块的行为

我们在 5.4 章节就随机创建了方块对象，各个方块在游戏中都有自己的行为，例如变化、左移、右移和快速下降。如果大方块进行变化，就需要从变化集合得到下一个变化，改变自己的 `pieces` 属性（表示当前的变化）。如果需要进行左移，就调用 `Piece` 对象的 `setSquaresXLocation` 方法让大方块对象的横坐标加上相应的值，右移就调用 `setSquaresXLocation` 方法让大方块对象的横坐标减去相应的值，`setSquaresXLocation` 方法已经在 5.4.3 中实现。如果大方块需要进行快速下降，就需要调用 `Piece` 对象的 `setSquaresYLocation` 的方法让 `Piece` 对象的纵坐标加上相应的值。

### 5.5.1 方块变化

在 5.4.2 中创建了大方块对象 (`Piece`)，在 `Piece` 中有一个 `changes` 的集合用来保存大方块的变化，因此在界面变化方块时，就可以拿到该集合的下一个元素，再设置当前的变化属性。为 `Piece` 对象新建一个 `change` 方法：

代码清单：code\tetris\src\org\crazyit\tetris\object\Piece.java

```
public void change(){
    if (this.changes.size() == 0) return;
    this.currentIndex += 1;
    //如果当前变化超过 changes 集合的大小，则从 0 开始变化
    if (this.currentIndex >= this.changes.size()) this.currentIndex = 0;
    this.squares = this.changes.get(this.currentIndex);
}
```



```
}
```

在 `change` 方法中, 先帮当前变化的索引加一, 再判断当前变化的索引是否超过变化集合 (`changes`) 的总数, 如果超过变化集合的总数, 则从 0 开始, 最后从变化中取得相应的元素设置到当前的 `squares` 属性中。由于在 `GamePanel` 的 `paint` 方法中, 我们使用 `Piece` 的 `squares` 属性来画界面的, 因此改变 `squares` 属性就可以达到改变方块形状的要求。

实现了 `change` 方法后, 为界面添加键盘事件。

代码清单: `code\tetris\src\org\crazyit\tetris\ui\MainFrame.java`

```
//添加键盘监听器
this.addKeyListener(new KeyAdapter() {
    public void keyPressed(KeyEvent e) {
        //键盘的上
        if (e.getKeyCode() == 38) change();
    }
});
```

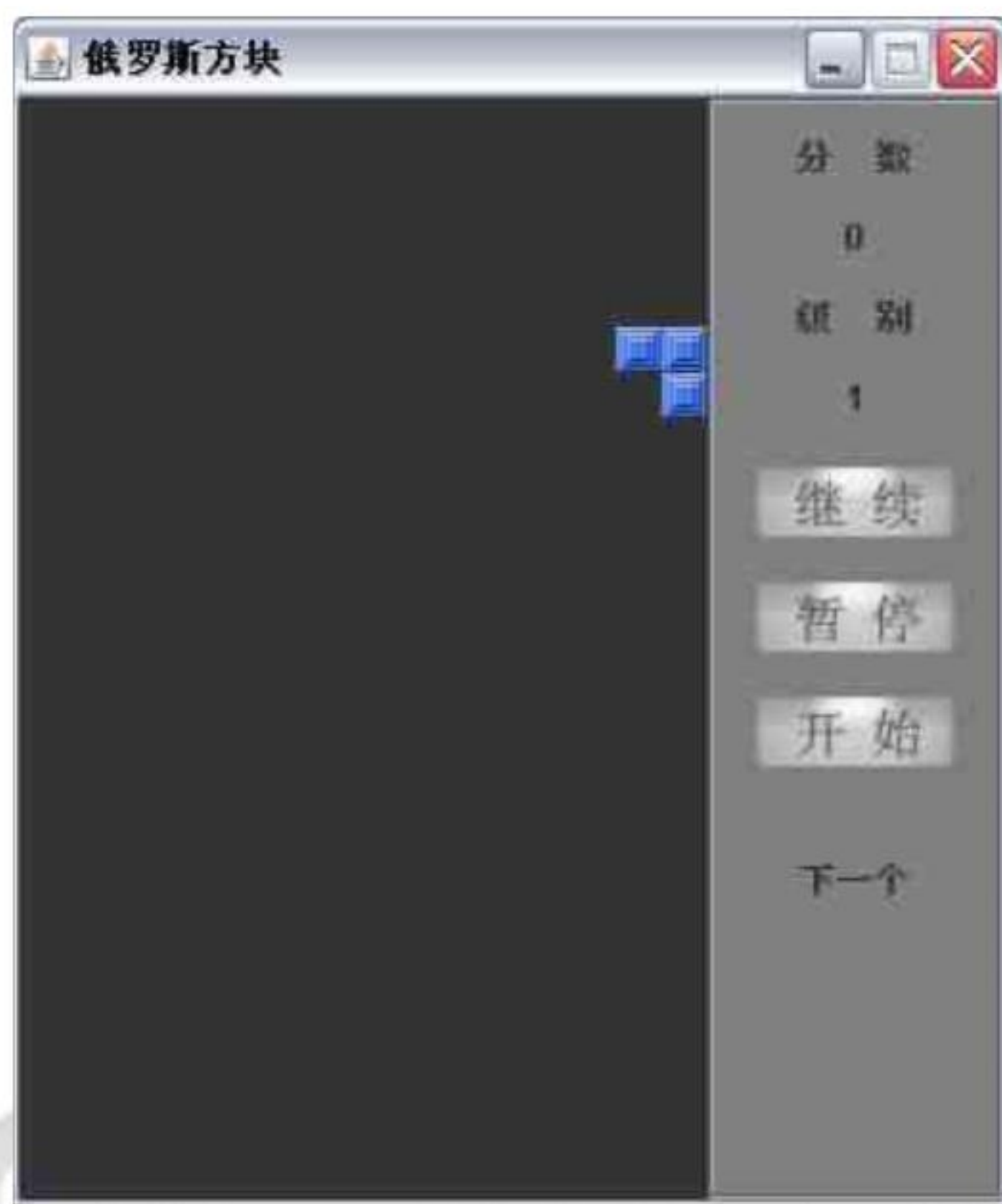
以下为 `MainFrame` 中的 `change` 方法的实现。 `MainFrame`:

```
//按键盘上时触发的方法
public void change() {
    if (this.currentPiece == null) return;
    this.currentPiece.change();
    this.gamePanel.repaint();
}
```

这样, 大方块的转换就实现了。但是, 还没有完全实现, 还会出现一些不理想的地方, 如图 5.9 和 5.10 所示。



图 5.9 大方块变化前



5.10 大方块变化后

当大方块在 **GamePanel** 的最右侧的时候，就会出现如图 5.10 的时的情况，同样地，如果大方块在 **GamePanel** 的最左边的时候，也会出现这种情况。因此，在转换的时候，需要作一些判断并作出相应的方块调整。为 **Piece** 对象添加两个方法，返回 **Piece** 对象的最小横坐标和最大横坐标的值。

代码清单：code\tetris\src\org\crazyit\tetris\object\Piece.java

```
//得到当前变化中 x 座标的最小值
public int getMinXLocation() {
    int result = Integer.MAX_VALUE;
    for (int i = 0; i < this.squares.size(); i++) {
        Square s = this.squares.get(i);
        if (s.getBeginX() < result) result = s.getBeginX();
    }
    return result;
}
//得到当前变化中 x 座标最大的值
public int getMaxXLocation() {
    int result = 0;
    for (int i = 0; i < this.squares.size(); i++) {
        Square s = this.squares.get(i);
        if (s.getBeginX() > result) result = s.getBeginX();
    }
    return result + SQUARE_BORDER;
}
```

由于 **Piece** 本身不保存横坐标和纵坐标的值，因此需要从当前变化（**squares** 属性）中得到横坐标最小或最大的那个 **Square** 对象，并返回这个 **Square** 对象的横坐标值，这个值就是整个大方块（**Piece**）



的最小（最大）横坐标。

然后对 `MainFrame` 中的 `change` 方法作出修改。

代码清单：code\tetris\src\org\crazyit\tetris\ui\MainFrame.java

```
//按键盘上时触发的方法
public void change() {
    if (this.currentPiece == null) return;
    this.currentPiece.change();
    //判断转换后左边是否越界
    //得到当前方块最小的 X 坐标
    int minX = this.currentPiece.getMinXLocation();
    //左边越界
    if (minX < 0) {
        //右移超过的部分
        this.currentPiece.setSquaresXLocation(-minX);
    }
    //判断转换后右边是否越界
    int maxX = this.currentPiece.getMaxXLocation();
    int gamePanelWidth = this.gamePanel.getWidth();
    //右边越界
    if (maxX > gamePanelWidth) {
        //左移超过 GamePanel 宽的部分
        this.currentPiece.setSquaresXLocation(-(maxX - gamePanelWidth));
    }
    this.gamePanel.repaint();
}
```

以上代码中的黑体部分就是新加的判断代码，如果左边越界，就将整个 `Piece` 对象右移超过的值；如果右边越界，就将整个 `Piece` 对象左移超过的值。左移与右移使用 `Piece` 中的 `setSquaresXLocation` 方法，这个方法已经在 5.4.2 中实现。加入了越界判断的代码后，再运行游戏可以看到具体的效果，已经不会再出现类似图 5.10 的情况了。

### 5.5.2 方块的左移和右移

在 5.4.2 中我们实现了 `setSquaresXLocation` 的方法，调用这个方法就可以改变整个 `Piece` 对象的横坐标值。下面实现方块的左移。

代码清单：code\tetris\src\org\crazyit\tetris\ui\MainFrame.java

```
//左，参数为距离
public void left(int size) {
    if (this.currentPiece == null) return;
    //判断是否已经在最左边边界
    if (this.currentPiece.getMinXLocation() <= 0) return;
    //得出移动距离
    int distance = -(Piece.SQUARE_BORDER * size);
    this.currentPiece.setSquaresXLocation(distance);
    this.gamePanel.repaint();
}
```

`MainFrame` 的 `left` 方法，`left` 方法提供一个 `size` 参数，表示移动的格数，再根据移动的格数乘以小方块的边长就可以得出移动的距离，最后调用 `Piece` 对象的 `setSquaresXLocation` 方法，就可以改变整个 `Piece` 对象的横坐标值。实现这些后，调用 `GamePanel` 的 `repaint` 方法重绘界面就可以实现左移。

右移的功能与左移一致，同样是计算出移动的距离，再设置 `Piece` 的横坐标，最后 `repaint` 即可。

以下是右移的具体实现。

代码清单: code\tetris\src\org\crazyit\tetris\ui\MainFrame.java

```
//右, 参数为距离(一格)
public void right(int size) {
    if (this.currentPiece == null) return;
    //判断是否超过 GamePanel 的宽
    if (this.currentPiece.getMaxXLocation() >= this.gamePanel.getWidth()) return;
    int distance = Piece.SQUARE_BORDER * size;
    this.currentPiece.setSquaresXLocation(distance);
    this.gamePanel.repaint();
}
```

无论左移还是右移, 都需要判断是否越界, 如果越界就直接返回, 不再往下执行。如果需要触发 left 和 right 方法, 需要加入键盘监听器。在 5.5.1 方块变换中, 已经加入了“上”的监听器。

代码清单: code\tetris\src\org\crazyit\tetris\ui\MainFrame.java

```
//添加键盘监听器
this.addKeyListener(new KeyAdapter() {
    public void keyPressed(KeyEvent e) {
        //上
        if (e.getKeyCode() == 38) change();
        //左
        if (e.getKeyCode() == 37) left(1);
        //右
        if (e.getKeyCode() == 39) right(1);
    }
});
```

### 5.5.3 方块下降

实现方块下降, 可以添加一个任务定时器实现, 即每隔多少时间就设置一下 Piece 对象的纵坐标, 然后再调用 GamePanel 的 repaint 方法。新建一个 TetrisTask 类处理时间间隔任务, 该类继承于 TimerTask, 为 TetrisTask 提供一个构造器, 并重写 run 方法。

代码清单: code\tetris\src\org\crazyit\tetris\ui\MainFrame.java

```
//主界面对象
private MainFrame mainFrame;
public TetrisTask(MainFrame mainFrame) {
    this.mainFrame = mainFrame;
}
public void run() {
    //得到当前正在运动的大方块
    Piece currentPiece = this.mainFrame.getCurrentPiece();
    //调用 Piece 的 setSquaresYLocation 方法
    currentPiece.setSquaresYLocation(Piece.SQUARE_BORDER);
    this.mainFrame.getGamePanel().repaint();
}
```

当游戏开始时, 就在 MainFrame 的 start 方法中启动这个任务调度器:

MainFrame 的 start 方法:

```
//初始化定时器
this.tetrisTask = new TetrisTask(this);
int time = 1000 / this.currentLevel;
```



```
this.timer.schedule(this.tetrisTask, 0, time);
```

以上代码启动任务调度器，需要注意的是，定时器运行间隔由级别决定，如果当前游戏的级别越高，则表示大方块的下降速度越快，以上的黑体代码实现了这一功能。方块不可能永远下降，当到达界面底部或者前面遇到障碍的时候，就需要停止下降（完成下降），这些将在下面章节实现。

#### 5.5.4 方块快速下降

方块的快速下降与正常下降实现一样，都是调用 **Piece** 的 **setSquaresYLocation** 方法实现。与正常下降一样，在这里暂时不实现停止下降（完成下降）。

代码清单：code\tetris\src\org\crazyit\tetris\ui\MainFrame.java

```
//下加速
public void down() {
    if (this.pauseFlag) return;
    if (this.currentPiece == null) return;
    int distance = Piece.SQUARE_BORDER;
    //调用 Piece 的方法，改变纵坐标的值
    this.currentPiece.setSquaresYLocation(distance);
    this.gamePanel.repaint();
}
```

当按着键盘中的“下”时，就可以执行以上的 **down** 方法，使用方块加速向下。

#### 5.5.5 判断是否停止下降

在一个大方块下降的过程中，如果该大方块到达了界面的底部或者遇到障碍，就需要停止下降并显示下一个大方块。判断一个大方块是否到了界面底部，需要得到大方块的最大纵坐标和界面的最高值。我们为 **Piece** 对象提供一个获取纵坐标最大值的方法。

代码清单：code\tetris\src\org\crazyit\tetris\object\Piece.java

```
//得到当前变化中 Y 座标最大值
public int getMaxYLocation() {
    int result = 0;
    for (int i = 0; i < this.squares.size(); i++) {
        Square s = this.squares.get(i);
        if (s.getBeginY() > result) result = s.getBeginY();
    }
    return result + SQUARE_BORDER;
}
```

从 **Piece** 当前变化的集合中得到纵坐标最大的那个 **Square** 对象，并返回它的纵坐标。那么在 **MainFrame** 中可以提供一个方法，判断 **Piece** 的最大纵坐标是否超过了界面（**GamePanel**）的高。

代码清单：code\tetris\src\org\crazyit\tetris\ui\MainFrame.java

```
//判断是否到界面最底部
public boolean isButtom() {
    return this.currentPiece.getMaxYLocation() >= this.gamePanel.getHeight();
}
```

#### 5.5.6 创建界面的二维数组

当一个 **Piece** 对象完成了下降，我们需要将这个 **Piece** 里面所有的 **Square** 对象记录在 **MainFrame**



中，我们为 **MainFrame** 提供一个二维数组，这个二维数组存放 **Square** 对象，每个完成下降后的 **Piece**，就将自己的所有 **Square** 对象存放到这个二维数组中。下面代码初始化一个 **Square** 的二维数组。

代码清单：code\tetris\src\org\crazyit\tetris\ui\MainFrame.java

```
//初始化界面二维数组
private void initSquares() {
    //得到宽可以存放的方块个数
    int xSize = this.gamePanel.getWidth()/Piece.SQUARE_BORDER;
    //得到高可以存放的方块个数
    int ySize = this.gamePanel.getHeight()/Piece.SQUARE_BORDER;
    //构造界面的二维数组
    this.squares = new Square[xSize][ySize];
    for(int i = 0; i < this.squares.length; i++) {
        for (int j = 0; j < this.squares[i].length; j++) {
            this.squares[i][j] = new Square(Piece.SQUARE_BORDER * i,
                Piece.SQUARE_BORDER * j);
        }
    }
}
```

这里需要注意的是，创建二维数组时，都会创建每一个 **Square** 对象，但是这些 **Square** 对象并没有对应的图片，当一个 **Piece** 完成下降的时候，才会给这些 **Square** 设置上相应的图片。

那么我们就需要去为 **GamePanel** 的 **paint** 方法加入绘画这个二维数组的代码。

代码清单：code\tetris\src\org\crazyit\tetris\ui\GamePanel.java

```
public void paint(Graphics g) {
    g.drawImage(this.background, 0, 0, this.getWidth(),
        this.getHeight(), null);
    Piece currentPiece = this.mainFrame.getCurrentPiece();
    ImageUtil.paintPiece(g, currentPiece);
    //绘画小方块的二维数组
    Square[][] squares = this.mainFrame.getSquares();
    if (squares == null) return;
    for (int i = 0; i < squares.length; i++) {
        for (int j = 0; j < squares[i].length; j++) {
            Square s = squares[i][j];
            if (s != null) {
                g.drawImage(s.getImage(), s.getBeginX(), s.getBeginY(), this);
            }
        }
    }
}
```

以上的黑体代码为新加的绘画小方块二维数组的代码，从 **MainFrame** 中得到二维数组，然后通过 **drawImage** 方法将每一个 **Square** 对象画到 **GamePanel** 中，由于 **Square** 对象中保存了图片、开始横坐标和纵坐标信息，因此 **paint** 方法不需要再做其他复杂的处理。

如果一个 **Piece** 在正常下降或者快速下降的时候，就需要判断是否到达了界面的最底部，如果到达了界面的最底部，就需要该 **Piece** 中所有的 **Square** 对象加入到界面的二维数组中。

**MainFrame** 中将 **Piece** 中的 **Square** 加入二维数组的方法：

```
//将 Piece 中所有的 Square 都加入到二维数组中
private void appendToSquares() {
    List<Square> squares = this.currentPiece.getSquares();
    //遍历 Piece 中的 Square
```



```

for(Square square : squares) {
    for(int i = 0; i < this.squares.length; i++) {
        for (int j = 0; j < this.squares[i].length; j++) {
            //得到当前界面中的 Square
            Square s = this.squares[i][j];
            //判断 Square 是否一致
            if (s.equals(square)) this.squares[i][j] = square;
        }
    }
}
this.gamePanel.repaint();
}

```

以上代码将当前的 **Piece** 对象中所有的 **Square** 加入到界面的二维数组中，需要注意的是黑体部分的代码，判断两个 **Square** 是否相等使用了 **equals** 方法，因此 **Square** 需要重写 **Object** 的 **equals** 方法，两个 **Square** 是否相等，判断的标准是两个 **Square** 的横坐标和纵坐标是否相等。修改 **down** 方法与定时器的 **run** 方法，当 **Piece** 下降到界面底部的时候，就调用上面的 **appendToSquares** 方法。此时再运行游戏，可以看到效果如图 5.11 所示。



图 5.11 大方块到达界面底部

### 5.5.7 判断是否遇到障碍

在 5.5.5 中实现了判断大方块是否到达界面的最底部，除了这个判断外，还需要判断方块下降的过程中是否遇到了障碍，也就是判断二维数组中的某个 **Square** 对象是否有图片。由于我们在创建二维数组的时候，这个数组里面存放的是一些 **Square** 对象，但是这些 **Square** 对象是没有 **image** 属性的

(MainFrame 的 initSquares 方法), 当一个 Square 被固定到某个位置的时候, 就在二维数组中查找相应的 Square 对象, 用刚下降的 Square 对象替换成原来二维数组中的 Square 对象。简单的说, 创建二维数组时, 每一个 Square 对象都是空的 (没有图片)。以下方法一个 Piece 下降的时候是否遇到障碍。

代码清单: code\tetris\src\org\crazyit\tetris\uiMainFrame.java

```
//判断当前的 Piece 是否遇到障碍, 返回 true 表示遇到障碍, 返回 false 表示没有遇到
public boolean isBlock() {
    List<Square> squares = this.currentPiece.getSquares();
    for (int i = 0; i < squares.size(); i++) {
        Square s = squares.get(i);
        //需要拿一个 Square 的最大 Y 座标
        Square block = getSquare(s.getBeginX(), s.getBeginY() + Piece.SQUARE_BORDER);
        //block 非空表示遇到障碍
        if (block != null) return true;
    }
    return false;
}
//根据开始座标在当前界面数组中查找相应的 Square
private Square getSquare(int beginX, int beginY) {
    for (int i = 0; i < this.squares.length; i++) {
        for (int j = 0; j < this.squares[i].length; j++) {
            Square s = this.squares[i][j];
            //与参数的开始座标相同并且图片不为空
            if (s.getImage() != null && (s.getBeginX() == beginX) &&
                (s.getBeginY() == beginY)) {
                return s;
            }
        }
    }
    return null;
}
```

当一个 Piece 正在下降的时候, 就到界面的二维数组中寻找对应的 Square 对象, 如果找到的 Square 对象没有图片, 那么在这次下降中就没有遇到障碍, 如果找到的 Square 对象有图片, 就证明遇到了障碍。遇到障碍后, 就调用 5.5.6 中的 appendToSquares 方法, 将当前的 Piece 对象中所有的 Square 加入到界面的二维数组中。实现了 isBlock 方法后, 在快速下降和正常下降的方法中加入 isBlock 的判断, 得到的效果如图 5.12 所示。



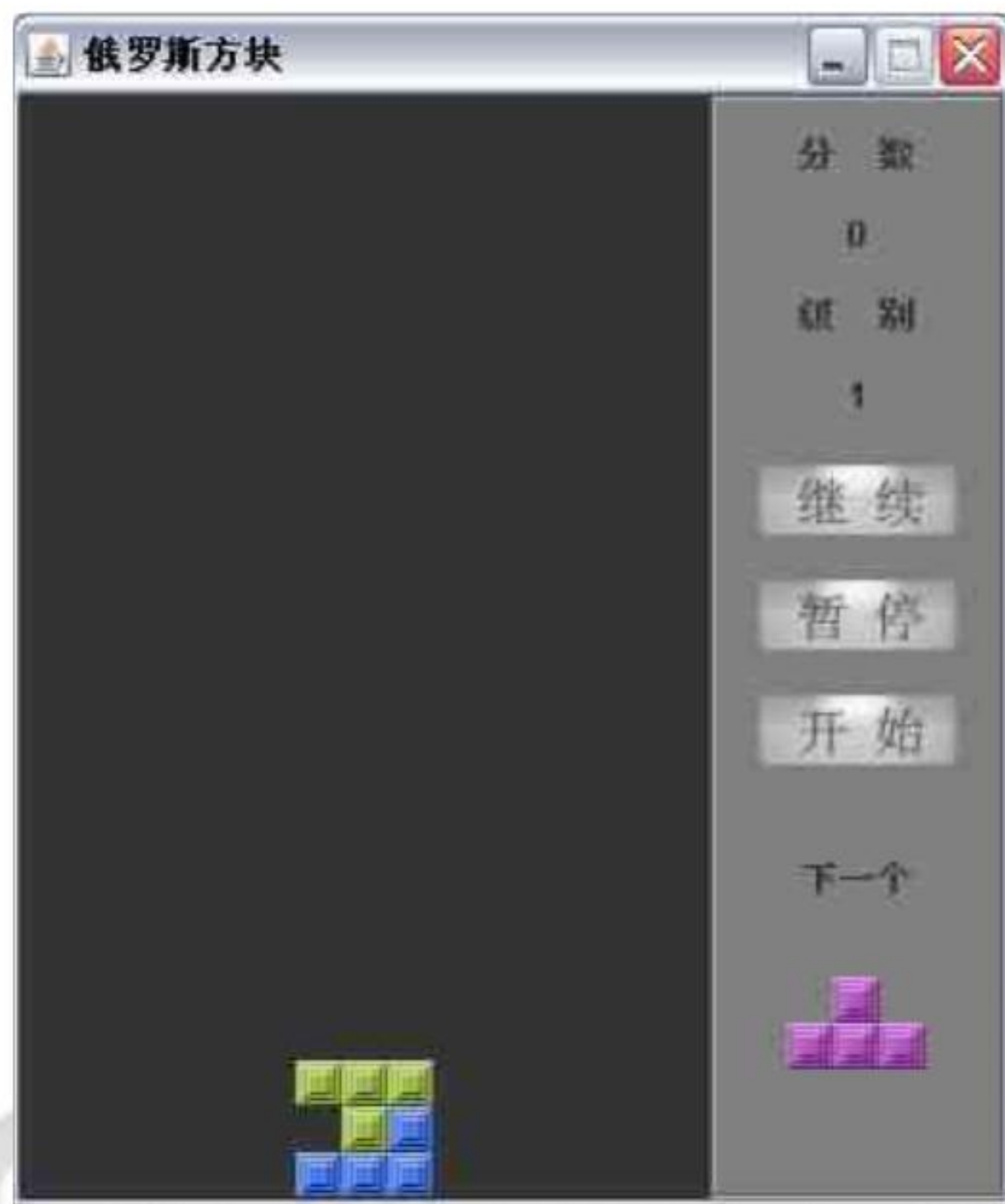


图 5.12 下降的方块遇到障碍

### 5.5.8 方块结束下降

当方块到达界面的最底或者遇到障碍，需要将当前方块设置为下一个方块，在 `MainFrame` 中我们已经提供了一个 `nextPiece` 的属性来代表下一个方块，那么只需要将下一个方块设置为当前方块就可以重新得到一个正在下降的方块，还需要调用 `PieceCreator` 来重新创建下一个的方块。

代码清单：`code\tetris\src\org\crazyit\tetris\uiMainFrame.java`

```
//创建下一个
private void createNextPiece() {
    this.nextPiece = this.creator.createPiece(NEXT_X, NEXT_Y);
    this.repaint();
}
//一个 Piece 对象完成下降
private void finishDown() {
    //将当前的 Piece 设置为下一个 Piece
    this.currentPiece = this.nextPiece;
    //设置新的 Piece 坐标
    this.currentPiece.setSquaresXLocation(-NEXT_X);
    this.currentPiece.setSquaresXLocation(BEGIN_X);
    this.currentPiece.setSquaresYLocation(-NEXT_Y);
    this.currentPiece.setSquaresYLocation(BEGIN_Y);
    //创建下一个 Piece
    createNextPiece();
}
```

如果一个方块在到达界面最底部或者遇到障碍时，就可以调用 `finishDown` 来执行下一个方块的下降与创建新的下一下方块。

## 5.6 消除行、计分与级别的提升

在 5.5 章节，我们已经实现了方块的下降，当方块结束下降的时候，就需要消除被全部填充小方块的行。当有方块行被消除时，就要加入一定的分数，如果分数到达一定的程序，就要提升游戏级别，游戏级别会影响方块的下降速度。

### 5.6.1 消除行

一个方块完成下降，就会将方块里面的小方块填充到界面的二维数组中，那么我们就可以对这个二维数组进行遍历，如果一行中都填充了小方块，那么就将该行所有的 `Square` 对象的图片设置为 `null`，这样就可以消除该行的显示。

代码清单：code\tetris\src\org\crazyit\tetris\ui\MainFrame.java

```
//得到可以清理行集合
private void cleanRows() {
    //使用一个集合保存被删除的行的索引
    List<Integer> rowIndexes = new ArrayList<Integer>();
    for (int j = 0; j < this.squares[0].length; j++) {
        int k = 0;
        for (int i = 0; i < this.squares.length; i++) {
            Square s = this.squares[i][j];
            //如果该格有图片，则 k+1
            if (s.getImage() != null) k++;
        }
        //如果整行都有图片，则消除
        if (k == this.squares.length) {
            //再次对该行进行遍历，设置该行所有格的图片为 null
            for (int i = 0; i < this.squares.length; i++) {
                Square s = this.squares[i][j];
                s.setImage(null);
            }
            rowIndexes.add(j);
        }
    }
}
```

以上的 `cleanRows` 方法消除全部被填充的行，当消除了这些方块行后，我们使用了一个集合来保存被删除行的索引（以上的黑体代码），这个集合将在下面的方法中使用。消除行的具体效果如图 5.13 所示。



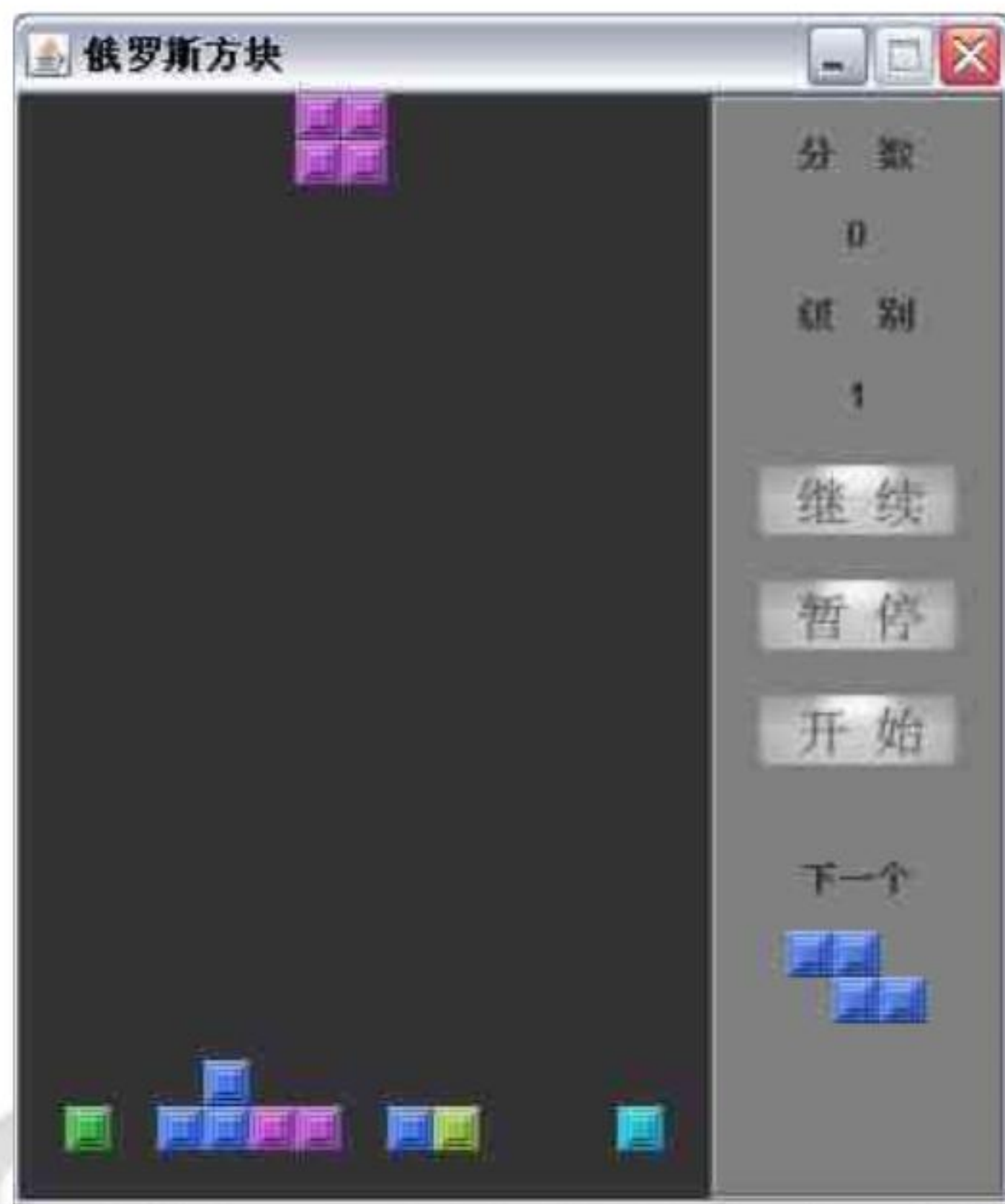


图 5.13 消除方块行

我们可以看到，当消除了一个方块行后，在该行上面的各个方块，仍然“飘浮”在空中，因此，在消除行后，我们还要对其他“飘浮”在空中的小方块进行处理，使它们能“下降”到相应的位置上。在 `cleanRows` 方法中，我们使用了一个集合来保存被删除的行的索引，因此可以提供一个方法，在删除行后，利用被删除行的索引来处理这些“飘浮”在空中的小方块。

代码清单：`code\tetris\src\org\crazyit\tetris\ui\MainFrame.java`

```
//处理行消除后其他 Square 的"下降", 参数为被删除的行的索引集合
private void handleDown(List<Integer> rowIndexes) {
    if (rowIndexes.size() == 0) return;
    //从被删除的行中拿出索引最小的行
    int minCleanRow = rowIndexes.get(0);
    int cleanRowSize = rowIndexes.size();
    //处理下降的 Square
    for (int j = this.squares[0].length - 1; j >= 0; j--) {
        if (j < minCleanRow) {
            //遍历上面的行，即悬浮的行
            for (int i = 0; i < this.squares.length; i++) {
                Square s = this.squares[i][j];
                if (s.getImage() != null) {
                    //得到下降前的图片
                    Image image = s.getImage();
                    s.setImage(null);
                    //得到下降后对应的 Square 对象，数组的二维值要加上消除行的行数
                    Square sdown = this.squares[i][j] + cleanRowSize;
                    sdown.setImage(image);
                }
            }
        }
    }
}
```

```

    }
    }
    }
    }
    }

```

以上的 `handleDown` 方法处理这些“飘浮”在空中的小方块，对界面的二维数组进行遍历，得到具体的某个“飘浮”在空中的小方块，得到这些小方块的图片后，使用一个 `Image` 对象来保存图片，再将这些小方块在二维数组中的二维索引加上消除的行数，即增大方块在二维数组中的二维值。

实现了 `handleDown` 方法后，就可以在消除方块行的方法最后调用 `handleDown` 方法：

```

//得到可以清理行集合
private void cleanRows() {
    //省略消除方块行的代码
    //处理悬浮的 Square
    handleDown(rowIndexes);
}

```

### 5.6.2 加入分数计算与级别提升

为俄罗斯方块加入分数计算十分简单，只要为 `MainFrame` 加入一个 `score` 的值，然后在每次清理了方块行后再加入相应的分数即可。如果分数到达一定值，就将级别提升。

代码清单：code\tetris\src\org\crazyit\tetris\uiMainFrame.java

```

//加入分数
private void addScore() {
    //加分
    this.score += 10;
    this.scoreLabel.setText(String.valueOf(score));
    //如果可以被 100 整除，则加一级
    if ((this.score % 100) == 0) {
        this.currentLevel += 1;
        this.levelLabel.setText(String.valueOf(this.currentLevel));
        //重新设置定时器
        this.timer.cancel();
        this.timer = new Timer();
        this.tetrisTask = new TetrisTask(this);
        //设置方块下降速度
        int time = 1000 / this.currentLevel;
        timer.schedule(this.tetrisTask, 0, time);
    }
}

```

以上的 `addScore` 方法，在 `cleanRows` 方法（消除方块行）中调用，消除了一行就加 10 分，如果分数为 100 的倍数，那么就将级别提升一级并重新设置定时器，游戏提高了一个级别，`MainFrame` 中的 `currentLevel` 属性就会增加，那么方块的下降速度将会提高。

## 5.7 游戏操作

在俄罗斯方块游戏中，会涉及一些简单的操作，例如游戏的暂停、继续和游戏失败的提示。游戏的开始我们在前面的章节中已经实现，本小节实现游戏的暂停与继续。



### 5.7.1 游戏的暂停

游戏中使用了一个定时器来处理方块的下降，如果游戏暂停的话，就需要将这个定时器停止，并且屏蔽其他操作（变化、快速下降、左移和右移）。可以在 **MainFrame** 中提供一个全局的布尔值来表示当前游戏的状态。

代码清单：code\tetris\src\org\crazyit\tetris\uiMainFrame.java

```
//暂停的标识, true 为暂停
private boolean pauseFlag = false;
//暂停游戏
public void pause() {
    this.pauseFlag = true;
    if (this.timer != null) this.timer.cancel();
    this.timer = null;
}
```

还需要为其他操作加入判断，如果当前是暂停的话，就不进行其他操作。

```
if (this.pauseFlag) return;
```

### 5.7.2 游戏继续

继续游戏，只需要重新创建定时器并执行 **TetrisTask**，但是需要作一些额外的判断，如何游戏没有开始或者正在游戏正在进行的时候，就不用再创建定时器。

代码清单：code\tetris\src\org\crazyit\tetris\uiMainFrame.java

```
//继续游戏
public void resume() {
    if (!this.pauseFlag) return;
    this.timer = new Timer();
    this.tetrisTask = new TetrisTask(this);
    int time = 1000 / this.currentLevel;
    timer.schedule(this.tetrisTask, 0, time);
    this.pauseFlag = false;
}
```

### 5.7.3 判断游戏失败

游戏失败的标准是界面最上方如果有方块的存在，那么游戏失败。在游戏界面中我们保存了一个二维数组来表示这些小方块，我们只需要遍历二维数组作出判断即可。

代码清单：code\tetris\src\org\crazyit\tetris\uiMainFrame.java

```
//判断是否失败, true 为失败, false 反之
private boolean isLost() {
    for (int i = 0; i < this.squares.length; i++) {
        Square s = this.squares[i][0];
        if (s.getImage() != null) return true;
    }
    return false;
}
```

注意：只需要对界面中最上面的行进行遍历，这样可以提升性能。



## 5.8 本章总结

本章编写了一个较为简单的俄罗斯方块，详细介绍了俄罗斯方块中的各个功能的实现，例如方块的创建、下降、消除等，并实现了计分与级别的功能，将方块堆砌界面抽象成为一个二维数组，并将每个小方块抽象成为一个 **Square** 对象，而一个正在下降的大方块由一个 **Piece** 对象来代表，方块的创建、下降和消除，都是通过操作对象来实现，讲述了如何随机获得方块图片，再由这些方块图片来随机创建大方块。本章中的俄罗斯方块相对较为简单，如果需要开发更复杂的俄罗斯方块，可能还包括自定义方块、自定义游戏规则等一系列复杂的功能，这些功能都可以由用户去配置，本章中的俄罗斯方块并不是为了开发多复杂的游戏，而是希望能通过这个简单的游戏，向大家展现这个游戏的实现原理。希望本章中这个简单的俄罗斯方块可以让大家充分了解这些基于 **swing** 的游戏原理，可以引导大家开发出更为强大与完善的游戏。



## 第6章 仿Windows画图

### 6.1 画图软件概述

我们平时所使用的图形处理工具有 PhotoShop、Windows 画图工具等，其中 PhotoShop 是一款非常强大的图形处理工具，Windows 画图工具则是一款较为简单的画图工具，功能较为简单，相信经常使用 Windows 系统的读者都比较熟悉，是一种比较简单与具有代表性的画图工具，虽然功能不够强大，但具有大多图片处理程序所必需的基本功能：铅笔画图、各种数学函数图形、填色、取色、橡皮擦等等功能。

本文将使用 Java 语言去实现 Windows 的大部分功能，包括取色、各种数学函数图形、橡皮擦、喷枪、颜色编辑等功能，除了这些绘图功能，还会实现打开图片、保存图片等文件操作功能。画图工具的最终效果如图 6.1 所示。

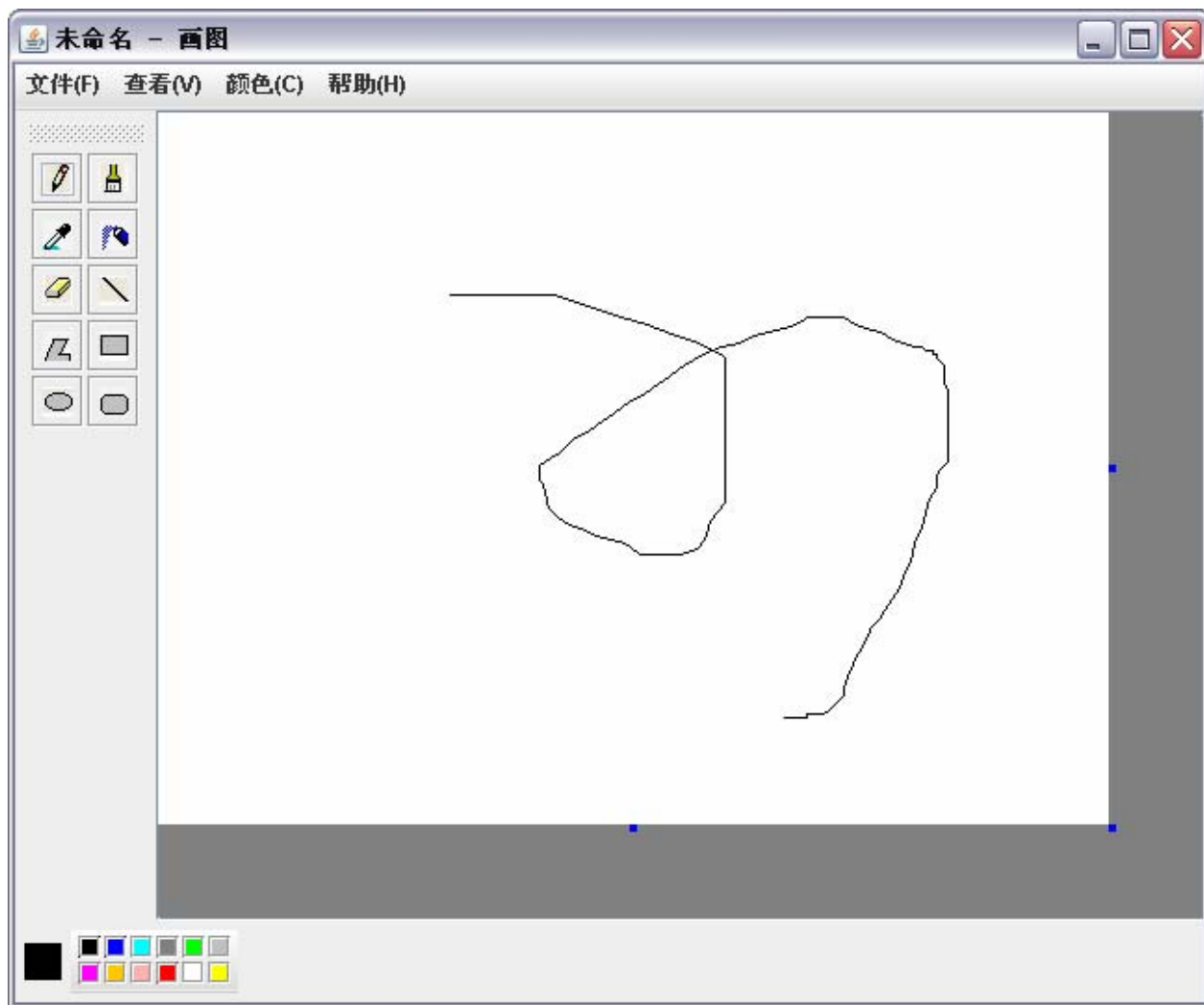


图 6.1 画图

## 6.2 画图工具原理

我们可以考虑一下如何使用 Java 程序去实现这些功能，如果需要进行画图，那么我们当然就需要 Graphics 类来向界面画上相应的内容，如果需要进行文件操作，我们可以使用 Java 的 IO 来实现。

### 6.2.1 画线

在 Graphics 中许多方法，其中有一个 drawLine 的方法，使用该方法我们可以将线画到界面中，该方法中有四个参数，分别是线的开始点坐标 (x、y) 与线的结束点的坐标 (x、y)，因此，如果需要调用该方法来画线的话，需要捕获用户在界面中按下鼠标的点坐标与放开鼠标时的点坐标。当调用了 drawLine 方法后，我们再对界面的组件进行一次 repaint 就可以实现画线的功能。

### 6.2.2 其他画图功能

画线我们可以调用 drawLine 方法，那么画椭圆的话可以调用 Graphics 类的 drawOval 方法，如果需要画矩形的话，可以调用 drawRect 方法。如果可以实现橡皮擦的功能，可以将鼠标经过的区域画上白色的线。实现喷涂的功能，可以在当前鼠标点击的区域中画上相应的点，Graphics 类中提供了一个 fillRect 的方法，我们可以利用该方法去填充当前的区域。除这些画图功能外，我们还需要提供一个刷子的功能，刷子功能可以看作是一个画笔功能，只是使用刷子画出来的线比画笔更粗而已。

### 6.2.3 保存图片功能

我们可以在画图的界面中保存一个 BufferedImage 的对象，那么可以通过这个对象得到一个 Graphics 对象，得到该对象，就可以使用它的 drawXXX 的方法来进行画图，到最后需要进行保存的时候，我们可以将这个 BufferedImage 对象通过 ImageIO 的 writer 方法写到文件中。

只要知道使用 Java 程序来实现画图的原理，实现程序就十分的简单，关键是如何计算各个工具的有效范围。

## 6.3 创建画图工具的各个对象

使用 Windows 的画图软件，发现在编辑图片的时候，有个相似的过程，首先是用鼠标选择需要使用的工具，然后就在画板中用鼠标进行拖动、点击等动作，画板会显示出相应工具的所产生的效果，所以在这里设计一个命名为 Tool 的接口，这个接口是所有工具的接口，里面定义了一系列的鼠标动作。实际上在这个画图工具中，所有的工具都必须遵守一定的规范，即使用鼠标进行拖动、点击等动作，当需要定义某些规范的时候，我们可以将这些规范写到一个接口中，那么这个接口所有的实现类都要遵守这个规范，这也是本章将工具作为一个接口的原因。

在本章中，画图软件的主界面使用 ImageFrame，该类继承于 JFrame，该类会初始化画图软件的各种组件。由于我们有选择打开图片文件的操作，所以会有一个扩展 javax.swing.JFileChooser 类（为选择文件提供一种简单的窗口选择机制）的 ImageFileChooser 类，用于处理选择文件时的过滤等操作。

因为绘图的功能已经全部由 Tool 的实现类去实现，所以除了绘图外的其它功能的逻辑实现，就全部放到 ImageService 类中，本章中的类图如图 6.2 所示。



- ❑ `static final String ROUNDRECT_TOOL` , 圆角矩形工具类型。
- ❑ `static final String ATOMIZER_TOOL`, 喷墨工具类型。
- ❑ `static final String COLORPICKED_TOOL` , 颜色选择工具类型。
- ❑ `void mouseDragged( MouseEvent e)`, 当捕捉到鼠标拖动时调用的方法定义。
- ❑ `void mouseMoved MouseEvent e)`, 当捕捉到鼠标移动时调用的方法定义。
- ❑ `void mouseReleased MouseEvent e)`, 当捕捉到鼠标松开时调用的方法定义。
- ❑ `void mousePressed MouseEvent e)`, 当捕捉到鼠标按下时调用的方法定义。
- ❑ `void mouseClicked MouseEvent e)`, 当捕捉到鼠标点击时调用的方法定义。

从接口中定义的属性与方法可以看出, 在接口中只定义工具的类型, 还有定义工具鼠标动作的方法, 就不再做任何的事情, 这些方法由它的实现类去具体实现。在某个对象中需要使用到 **Tool** 的实现类时, 我们可以使用一个 **ToolFactor** 的类来得到具体的某个 **Tool** 实现类, **ToolFactory** 返回的都是 **Tool** 接口, 因此使用者根本不需要关心使用的是哪一个实现类, 当代码发生改变的时候, 也可以减少代码的修改。换言之, 使用者只与 **ToolFactory** 耦合。

### 6.3.2 Tool的实现类AbstractTool

**AbstractTool** 是 **Tool** 的实现类, 也是一个抽象类, 所以并不能被创建, 只能被继承。此类实现 **Tool** 中定义的所有方法, 并扩展了其它方法, 让其子类继承或者重写。该类中为其他的工具类提供了大部分的实现, 那么它的子类就可以不必再做重复的实现, 只关心与本类相关的逻辑, **AbstractTool** 所定义的方法如下:

- ❑ `AbstractTool( ImageFrame frame )`, 让子类调用的构造器, 以 **ImagerFrame** 为参数, 用于获取画板的属性。
- ❑ `AbstractTool( ImageFrame frame, String path )`, 让子类调用的构造器, 以 **ImagerFrame** 为参数, `path` 是工具的图标路径。
- ❑ `Cursor getDefaultCursor()`, 此方法获取默认鼠标指针的形状。
- ❑ `void setDefaultCursor( Cursor cursor )`, 设置鼠标指针指针, 以 **Cursor** 为参数。
- ❑ `void setPressX(int x)`, 设置鼠标按下的 `x` 坐标, `int` 类型的 `x` 为鼠标的 `x` 坐标。
- ❑ `void setPressY(int y)`, 设置鼠标按下的 `y` 坐标, `int` 类型的 `y` 为鼠标的 `y` 坐标。
- ❑ `int getPressX()`, 返回上次鼠标按下的 `x` 坐标。
- ❑ `int getPressY()`, 返回上次鼠标按下的 `y` 坐标。
- ❑ `void mouseDragged( MouseEvent e)`, 实现当捕捉到鼠标拖动时调用的方法。
- ❑ `void mouseMoved MouseEvent e)`, 实现当捕捉到鼠标移动时调用的方法。
- ❑ `void mouseReleased MouseEvent e)`, 实现当捕捉到鼠标松开时调用的方法。
- ❑ `void mousePressed MouseEvent e)`, 实现当捕捉到鼠标按下时调用的方法。
- ❑ `void mouseClicked MouseEvent e)`, 实现当捕捉到鼠标点击时调用的方法。
- ❑ `void createShape( MouseEvent e ,Graphics g )`, 画图形, 通过参数 `e` 去获取鼠标的轨迹, 并用 **Graphics** 类型的对象 `g` 去画图形。
- ❑ `void draw(Graphics g, int x1, int y1, int x2, int y2)`, 画图形, `g` 是用来画图形的对象, (`x1`, `y1`) 是起点坐标, (`x2`, `y2`) 是终点坐标。这个方法是一个空的方法, 主要是由其子类实现。
- ❑ `void dragBorder( MouseEvent e )`, 拖动边界, 也就是改变画布的大小。

接口用于定义规范, 那么抽象类就是用于实现部分的规范。当我们在编写程序的过程中发现, 有一类对象都必须遵守某些行为, 那么我们可以将这些行为都当作规范, 写到接口中; 如果有些对象实现了部分的行为, 其他的行为更希望让它的子类去实现, 那么我们可以将这些对象作为一个抽象类。

### 6.3.3 AbstractTool的子类

AbstractTool 一共有 ArrowTool (箭头)、PencilTool (铅笔)、BrushTool (刷子)、EraserTool (橡皮擦)、LineTool (直线)、RectTool (矩形)、PolygonTool (多边形)、RoundTool (椭圆形)、RoundRectTool (圆矩形)、AtomizerTool (喷墨)、ColorPickedTool (颜色选择) 11 个子类, 这些子类都是根据自己的情况重写 AbstractTool 的部分或者全部方法。由于在本设计中, 想这些类在外表现为 Tool 接口, 不希望被直接实例化, 所以此类的构造器私有, 并提供一个静态的方法获取 Tool 类型的此类实现, 如下:

❑ static Tool getInstance( ImageFrame frame ), 获取 Tool 类型的本类实例。

由于我们并不希望外界可以直接使用 new 关键字来创建这些类的实例, 因此在这里使用了单态模式, 所有的子类都提供了 getInstance 的方法来返回本类的实例, 并且所有的构造器都是私有的。在下面的章节中, 将会讲解如何实现这 11 个子类。

### 6.3.4 界面类ImageFrame

这个画图工具的界面的主要放在这个类中实现, 此类有以下方法:

- ❑ void init(), 设置化主界面。
- ❑ JPanel getDrawSpace(), 获取画布。
- ❑ JPanel getColorPanel(), 获取颜色面板。
- ❑ MyImage getBufferedImage(), 获取画板中的图片。
- ❑ void setBufferedImage( MyImage bufferedImage), 设置画板图片, MyImage 是 BufferedImage 的一个扩展类。
- ❑ void setTool( Tool tool ), 设置正在使用的工具。
- ❑ Tool getTool(), 获取正在使用的工具。
- ❑ JColorChooser getColorChooser(), 获取颜色选择器。
- ❑ JPanel createColorPanel(), 创建一个简单的颜色选择面板。
- ❑ JPanel getCurrentColorPanel(), 获取颜色选择面板。
- ❑ Dimension getScreenSize(), 获取 Dimension 类形 screenSize, screenSize 主要用于获取画板的高与宽等属性。
- ❑ void createMenuBar(), 创建文件、查看、颜色、帮助等菜单栏。
- ❑ JPanel createDrawSpace(), 创建画板。
- ❑ JPanel createToolPanel(), 创建用于画图的工具栏。

另外, 此类有一个继承 JPanel 的内部类 DrawSpace, 用于充当画图工具的画板, 此内部类只有一个方法, 就是一个用于绘图的方法, 如下:

❑ void paint( Graphics g ), 画图。

界面类类似于我们 MVC 模式中的 V (视图), 该类并不负责处理任何的逻辑, 主要负责从界面接收数据, 再传递给具体的业务类, 让其进行相关的处理。在本章, 负责处理画图功能的主要是 Tool 的实现类。

### 6.3.5 业务逻辑类ImageService

除鼠标的画图功能外 (画图功能由 Tool 的实现类完成), 初始化画板、图片的新建打开与保存、各种面板的显示与隐藏、颜色的编辑、整个界面的刷新、菜单等业务逻辑都放在这个类中实现, 该类包含了以下的方法:

❑ initDrawSpace( ImageFrame frame ), 初始化画板。

- ❑ Dimension getSize(), 获取屏幕的分辨率。
- ❑ repaint( Graphics g, BufferedImage bufferedImage ), 刷新界面。
- ❑ static Cursor createCursor( String path ), 创建鼠标图形。Path 是鼠标图形的路径。
- ❑ void save( boolean b, ImageFrame frame ), 保存图片。
- ❑ void open( ImageFrame frame ), 打开图片。
- ❑ void createGraphics( ImageFrame frame ), 创建新图片并初始化。
- ❑ void editColor( ImageFrame frame ), 编辑颜色。
- ❑ void exit( ImageFrame frame ), 退出画图软件。
- ❑ void menuDo( ImageFrame frame, String cmd ), 处理菜单事件。

除了画图功能外, ImageService 负责了整个画图工具的其他功能, 在本章中, 该类是无状态的 Java 对象, 它并没有保存一些状态属性。

### 6.3.6 文件选择类 ImageFileChooser

ImageFileChooser 类继承了 JFileChooser 类, JFileChooser 是 Java 提供的一个简单的文件选择机制, 我们这里扩展这个类, 是为了增加我们自己的文件过滤器。见以下方法:

- ❑ String getSuf(), 获取文件的后缀名。
- ❑ void addFilter(), 增加文件过滤器, 这里只选择图片类型的文件。

这个类中有一个继承 FileFilter 类的内部类 MyFileFilter, 这个内部类主要是重写 FileFilter 的 accept 方法, 判断是否是合法的文件类型, 如下:

- ❑ boolean accept( File f ), 判断是否是合法的文件类型。

在本小节中, 我们主要确定了画图工具所涉及的几个对象, 并定义了他们的行为与属性, 在下面章节中, 我们只要按照这些定义好的方法, 逐步去实现我们的画图工具。

## 6.4 主界面实现

在这个软件中, 主界面主要由左边的工具栏、下面的颜色选择板、占大部分区域的画图区、菜单等几部分组成, 用 BorderLayout 的排版方式, 左边工具栏在 BorderLayout.WEST 位置, 画图区在 BorderLayout.CENTER 位置, 颜色选择面板在 BorderLayout.SOUTH 位置。先看主界面的初始化:

### 6.4.1 初始化界面 (init()方法)

首先, 设置 JFrame 窗口的标题, 接下来初始化画图区域, 初始化为白色, 然后再获取 PENCIL\_TOOL(铅笔)类型的 Tool, 创建各种鼠标监听器, 并在监听的执行方法中调用 Tool 的相应方法, 最后获取左边工具栏面板、下面菜单栏面板、菜单, 并把这些面板与画图获取加到 JFrame 中。见以下代码。

代码清单: code\image\src\org\crazyit\image\ImageFrame.java

```
public void init() {  
    //设置标题  
    this.setTitle( "未命名 - 画图" );  
    //初始化画图  
    service.initDrawSpace( this );  
    //设置标题  
    //获取正在使用的工具
```

```

tool = ToolFactory.getToolInstance( this, PENCIL_TOOL );
//创建鼠标运动监听器
MouseMotionListener motionListener = new MouseMotionAdapter() {
    //拖动鼠标
    public void mouseDragged(MouseEvent e) {
        tool.mouseDragged( e );
    }
    //移动鼠标
    public void mouseMoved(MouseEvent e) {
        tool.mouseMoved( e );
    }
};
//创建鼠标监听器
MouseListener mouseListener = new MouseAdapter(){
    //松开鼠标
    public void mouseReleased( MouseEvent e ) {
        tool.mouseReleased( e );
    }
    //按下鼠标
    public void mousePressed(MouseEvent e) {
        tool.mousePressed( e );
    }
    //点击鼠标
    public void mouseClicked(MouseEvent e) {
        tool.mouseClicked( e );
    }
};
drawSpace.addMouseMotionListener( motionListener );
drawSpace.addMouseListener( mouseListener );
createMenuBar();
//以 drawSpace 为 viewport 去创建一个 JScrollPane
scroll = new JScrollPane( drawSpace );
//设置 viewport
ImageService.setViewport( scroll, drawSpace
    , bufferedImage.getWidth(), bufferedImage.getHeight() );
//将 panel 加到本 Frame 上面
this.add( scroll, BorderLayout.CENTER );
//this.add( toolPanel, BorderLayout.WEST );
//this.add( colorPanel, BorderLayout.SOUTH );
}

```

可以看到,这里有两种鼠标监听器, `MouseMotionListener` 和 `MouseListener`, `MouseMotionListener` 主要是监听鼠标的运动动作,我们实现了它的 `mouseDragger` (鼠标拖动)与 `mouseMoved` (鼠标移动)方法, `MouseListener` 负责监听鼠标的其它动作,我们实现了它的 `mouseReleased` (松开鼠标)、`mousePressed` (按下鼠标)和 `mouseClicked` (点击鼠标)三个方法。以上代码的黑体部分,这三行代码分别创建菜单、画图工具栏与颜色选择面板,如何创建我们将在 6.4.3、6.4.4 和 6.4.5 中详细描述。现在运行画图工具,可以看到效果如图 6.3 所示。



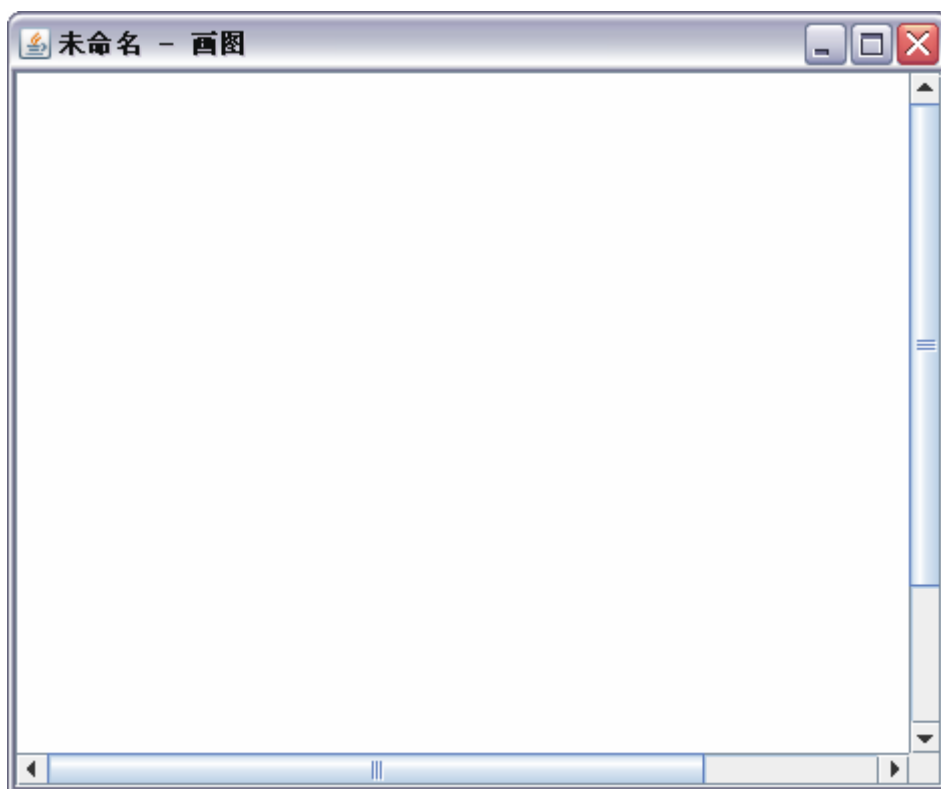


图 6.3 主界面

### 6.4.2 获取画板

这是一个画图工具, 所以需要一个可以绘图的区域, 在这里我们用继承 `JPanel` 的内部类 `DrawSpace` 去充当这个绘图区域, 见以下代码。

代码清单: `code\image\src\org\crazyit\image\ImageFrame.java`

```
// 画图区域
public class DrawSpace extends JPanel {
    /**
     * 重写 void paint( Graphics g )方法
     *
     * @param g Graphics
     * @return void
     */
    public void paint( Graphics g ) {
        //draw
        service.repaint( g, bufferedImage );
    }
}
```

从上面代码可以看到, 这个内部类比较简单, 只是继承 `JPanel`, 并重写 `JPanel` 的 `paint` 方法, 这里需要注意的是, 要调用此方法, 并不是直接调用 `paint` 方法, 而是调用 `ImageService` 的 `repaint` 方法。而获取这个画板就是去创建一个这个画板类的实例, 由于我们的画图软件是每次都只编辑一张图片, 所以这个创建画板的方法在本类中只被调用一次。首先是 `new` 一个 `DrawSpace` 实例, 再设置这个 `DrawSpace` 的大小, 并且返回, 见以下代码。

代码清单: `code\image\src\org\crazyit\image\ImageFrame.java`

```
//创建画板
public JPanel createDrawSpace() {
    JPanel drawSpace = new DrawSpace();
    //设置 drawSpace 的大小
    drawSpace.setPreferredSize(
        new Dimension( (int)screenSize.getWidth()
            , (int)screenSize.getHeight() - 150 ) );
    return drawSpace;
}
```

### 6.4.3 创建菜单

这个软件的菜单组织形式如下：

- 文件 (F)
  - 新建 (N)
  - 打开 (O)
  - 保存 (S)
  - 退出 (X)
- 查看 (V)
  - 工具箱 (T)
  - 颜料盒 (C)
- 颜色 (C)
  - 编辑颜色
- 帮助 (H)
  - 帮助主题
  - 关于

由于菜单比较简单,我们把文件、查看、颜色、帮助四个菜单文字放在一个 **String** 类型的数组 **menuArr** 里面,并迭代这个数组去创建一个 **JMenu**, **JMenu** 就是指菜单。同样,把他们下面的各个菜单项文字也放在一个 **String** 类型的二维数组里面,去迭代创建每个 **JMenuItem** (菜单项),每创建完一个,就为它加上一个动作监听器,去监听这个菜单项是否被点击。请看以下代码。

代码清单: code\image\src\org\crazyit\image\ImageFrame.java

```
//创建菜单
public void createMenuBar() {
    //创建一个 JMenuBar 放置菜单
    JMenuBar menuBar = new JMenuBar();
    //菜单文字数组, 与下面的 menuItemArr 一一对应
    String[] menuArr = { "文件(F)", "查看(V)", "颜色(C)", "帮助(H)" };
    //菜单项文字数组
    String[][] menuItemArr = {
        {"新建(N)", "打开(O)", "保存(S)", "-", "退出(X)"},
        {"工具箱(T)", "颜料盒(C)"},
        {"编辑颜色"},
        {"帮助主题", "关于"}
    };
    //遍历 menuArr 与 menuItemArr 去创建菜单
    for( int i = 0 ; i < menuArr.length ; i++ ) {
        //新建一个 JMenu 菜单
        JMenu menu = new JMenu( menuArr[i] );
```

```
for( int j = 0 ; j < menuItemArr[i].length ; j++ ) {  
    //如果 menuItemArr[i][j]等于 "-"  
    if ( menuItemArr[i][j].equals( "-" ) ){  
        //设置菜单分隔  
        menu.addSeparator();  
    } else {  
        //新建一个 JMenuItem 菜单项  
        JMenuItem menuItem = new JMenuItem( menuItemArr[i][j] );  
        menuItem.addActionListener( menuListener );  
        //把菜单项添加到 JMenu 菜单里面  
        menu.add( menuItem );  
    }  
}  
//把菜单添加到 JMenuBar 上  
menuBar.add(menu);  
}  
//设置 JMenuBar  
this.setJMenuBar( menuBar );  
}
```

以上代码的粗体部分，是为菜单添加相应的分隔符。在一般的下拉菜单中，可以通过分隔符将菜单划分为多个块。加入菜单后主界面的具体效果如图 6.4 所示。

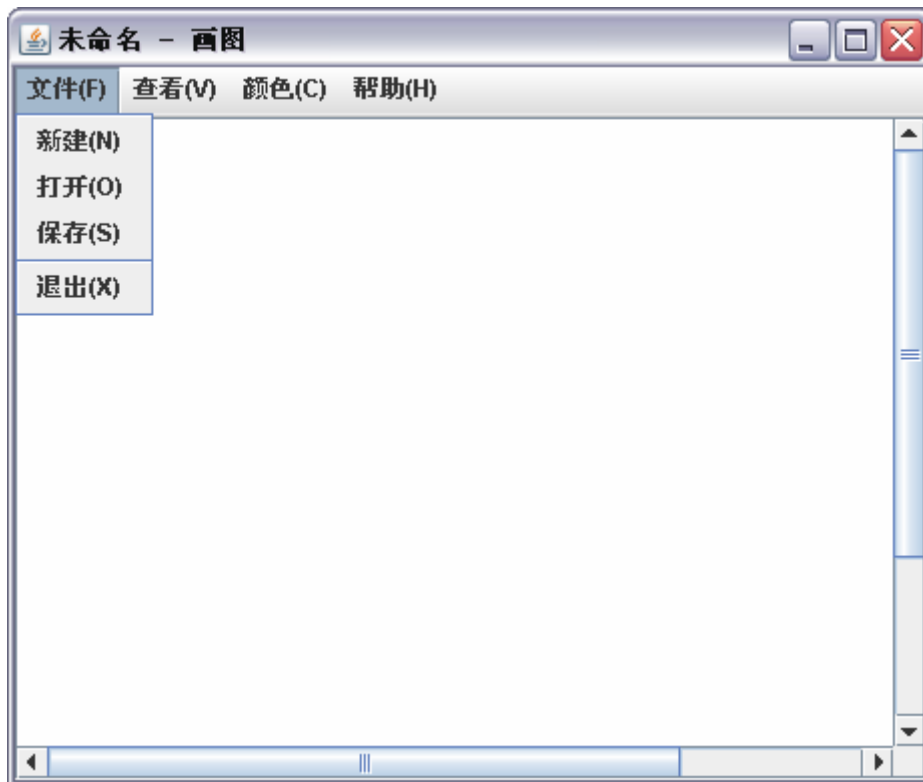


图 6.4 加入菜单后的界面

#### 6.4.4 创建画图工具栏

画图工具栏，这里实现的工具有铅笔、刷子、拾色器、喷枪、橡皮擦、直线、多边形、矩形、椭圆

形和圆矩形，这里，每个工具我们用一个 `JButton` 去代表它，使用 `JButton` 的 `JButton(AbstractAction)` 构造器去创建 `JButton`，用这个构造器创建 `JButton`，可以在 `AbstractAction` 中加入按键的图标，以图形的方式创建按钮，另外，我们会有一个 `AbstractAction` 的实现类。见以下代码：

代码清单：code\image\src\org\crazyit\image\ImageFrame.java

```
public JPanel createToolPanel() {
    //创建一个 JPanel
    JPanel panel = new JPanel();
    //创建一个标题为"工具"的工具栏
    JToolBar toolBar = new JToolBar( "工具" );
    //设置为垂直排列
    toolBar.setOrientation( toolBar.VERTICAL );
    //设置为可以拖动
    toolBar.setFloatable( true );
    //设置与边界的距离
    toolBar.setMargin( new Insets( 2, 2, 2, 2 ) );
    //设置布局方式
    toolBar.setLayout( new GridLayout( 5, 2, 2, 2 ) );
    //工具数组
    String[] toolarr = { PENCIL_TOOL, BRUSH_TOOL, COLORPICKED_TOOL
        , ATOMIZER_TOOL, ERASER_TOOL, LINE_TOOL, POLYGON_TOOL
        , RECT_TOOL, ROUND_TOOL, ROUNDRECT_TOOL };
    for( int i = 0 ; i < toolarr.length ; i++ ) {
        ImageAction action = new ImageAction(
            new ImageIcon("img/" + toolarr[i] + ".jpg")
            , toolarr[i], this );
        //以图标创建一个新的 button
        JButton button = new JButton( action );
        //把 button 加到工具栏中
        toolBar.add(button);
    }
    panel.add( toolBar );
    //返回
    return panel;
}
```

首先，我们创建一个放置这些按钮的 `JToolBar`，`JToolBar` 是 Java Swing 提供的一个工具栏类，并设置 `JToolBar` 的标题、排列方式（垂直）、是否可以拖动、与边界的距离、还有布局方式，接下来遍历存放这些工具类形的数组，每次都以工具图标的路径、工具名去创建一个 `ImageAction`（`AbstractAction`）的子类，并以这个 `ImageAction` 去创建一个 `JButton`，并把 `JButton` 加到 `JToolBar` 中。那么，怎么监听这些按钮？在这里，监听的工作就放到继承 `AbstractAction` 的 `ImageAction` 去实现，见 `ImageAction` 中重写的 `actionPerformed` 方法，这个方法就是用来执行监听到按钮被点击后的方法。

代码清单：code\image\src\org\crazyit\image\ImageAction.java

```
public void actionPerformed( ActionEvent e ) {
    //设置 tool
    tool = name != "" ? ToolFactory.getToolInstance( frame, name )
        : tool;
    if( tool != null ) {
        //设置正在使用的 tool
        frame.setTool( tool );
    }
}
```

```
if( color != null ) {  
    //设置正在使用的颜色  
    AbstractTool.color = color ;  
    colorPanel.setBackground( color );  
}  
}
```

代码中加粗的地方是这个方法的主要业务，首先能过工具的名字 **name** 去获取一个 **Tool** 接口的实现类实例，如果这个实例不为空，就把它设置为现在正使用的工具。而设置正在使用的颜色这段代码，是用于在监听颜色选择时用到，如果 **color** 不为空，就把当前的颜色设置为被选择的颜色。加入了画图工具栏后，具体的效果如图 6.5 所示。

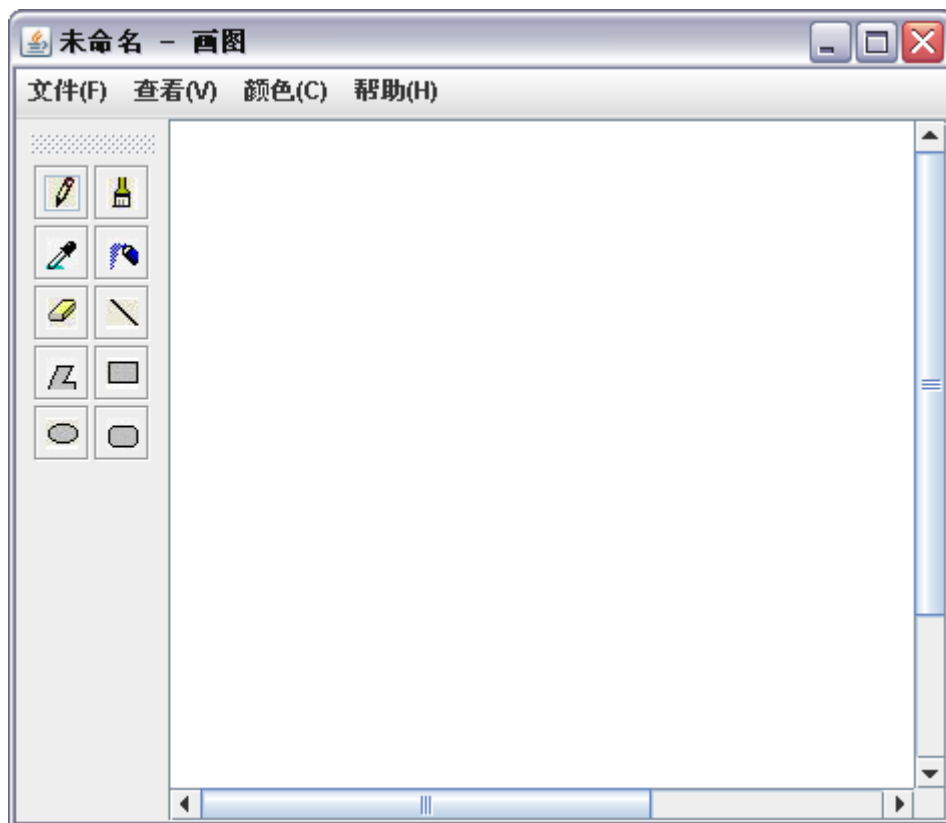


图 6.5 加入画图工具栏

### 6.4.5 创建颜色选择面板

这是一个简单的颜色选择面板，只有最基本的几种颜色选择（BLACK、BLUE、CYAN、GRAY、GREEN、LIGHT\_GRAY、MAGENTA、ORANGE、PINK、RED、WHITE、YELLOW），创建过程与创建工具栏的过程类似，用设置了颜色的按钮去代表这些颜色，首先创建一个 **JToolBar** 去放置这些颜色按钮，并设置这个 **JToolBar** 的布局方式、标题、是否可以拖动等属性，最后去遍历保存这些颜色类型的数组，每次都新建一个 **ImageAction**，并以这个 **ImageAction** 去创建一个 **JButton**，同时设置这个 **JButton** 的颜色，最后加到 **JToolBar** 中。见以下代码。

代码清单：code\image\src\org\crazyit\image\ImageFrame.java

```
public JPanel createColorPanel() {  
    //新建一个 JPanel  
    JPanel panel = new JPanel();
```

```
//设置布局方式
panel.setLayout( new FlowLayout( FlowLayout.LEFT ) );
//新建一个 JToolBar
JToolBar toolBar = new JToolBar( "颜色" );
//设置为不可拖动
toolBar.setFloatable( false );
//设置与边界的距离
toolBar.setMargin( new Insets( 2, 2, 2, 2 ) );
//设置布局方式
toolBar.setLayout( new GridLayout( 2, 6, 2, 2 ) );
//Color 类中的已有颜色
Color[] colorArr = { BLACK, BLUE, CYAN, GRAY, GREEN, LIGHT_GRAY
    , MAGENTA, ORANGE, PINK, RED, WHITE, YELLOW};
JButton[] panelArr = new JButton[ colorArr.length ];
//正在使用的颜色
currentColorPanel = new JPanel();
currentColorPanel.setBackground( Color.BLACK );
currentColorPanel.setPreferredSize( new Dimension( 20, 20 ) );
//创建这些颜色的 button
for( int i = 0 ; i < panelArr.length ; i++ ) {
    //创建 JButton
    panelArr[i] = new JButton( new ImageAction( colorArr[i], currentColorPanel ) );
    //设置 button 的颜色
    panelArr[i].setBackground( colorArr[i] );
    //把 button 加到 toolbar 中
    toolBar.add( panelArr[i] );
}
panel.add( currentColorPanel );
panel.add( toolBar );
//返回
return panel;
}
```

以上代码主要用于创建颜色选择面板，以上代码的黑体部分，使用了 **JButton** 来作为一种颜色的选择按钮，这些颜色选择按钮使用了 **ImageAction** 作为构造参数，**ImageAction** 在 6.4.4 中创建，该 **Action** 类负责处理用户点击工具栏或者选择颜色时的行为，现在可以运行画图工具，具体的效果如图 6.6 所示。

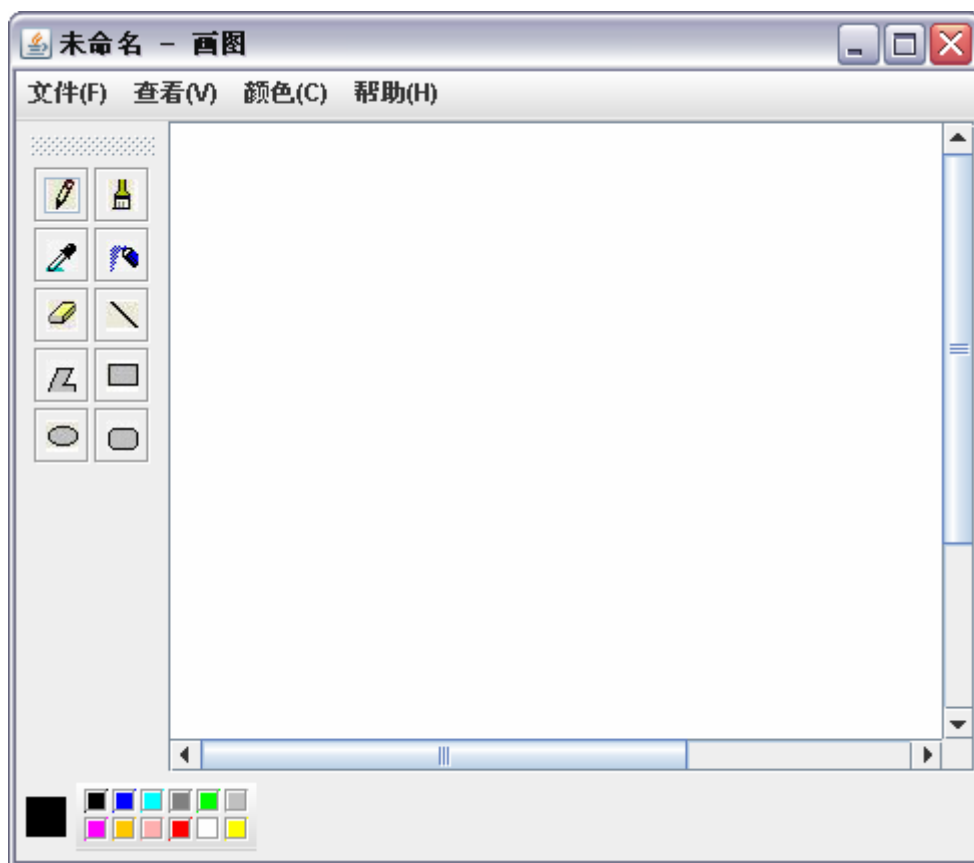


图 6.6 主界面

在本小节中，我们使用 **Swing** 创建了画图工具的主界面，主界面主要包括菜单、画图工具栏、颜色选择面板与画图区域等。接下来，我们将逐步实现画图里面的功能。

## 6.5 工具实现

前面说到，这里的工具类设计是有一个定义了工具所有方法的接口 **Tool**，这个接口有一个实现类 **AbstractTool**，该类实现了 **Tool** 接口的所有方法，并定义了一些方法供子类重写，**Tool** 接口、**AbstractTool** 类与它的子类关系，如图 6.7 所示。

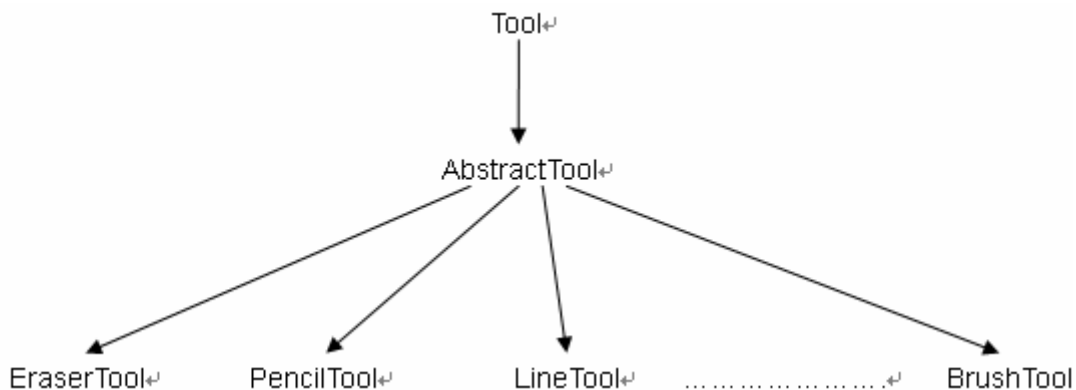


图 6.7 Tool 继承关系图

Tool 提供 `mouseDragged`、`mouseMoved`、`mouseReleased`、`mousePressed`、`mouseClicked` 五个方法接口，由其实现类实现，接下来先了解所有工具类的父类：`AbstractTool`。

### 6.5.1 实现拖动边框改变画布大小

在画图工具中，如果用户将鼠标移到画布边缘的时候，鼠标指针需要变成可拖动的图标，当按下鼠标左键时，就可以通过鼠标的移动来改变画布的大小。我们在创建主界面的时候，就将画布设计为一个 `JScrollPane`，因此对画布进行拖动，就可以设置画布的大小。在 `AbstractTool` 中提供一个 `dragBorder` 的方法。

代码清单：`code\image\src\org\crazyit\image\tool\AbstractTool.java`

```
//拖动画布边框
public void dragBorder( MouseEvent e ) {
    getFrame().getBufferedImage().setIsSaved( false );
    //获取鼠标现在的 x 与 y 坐标
    int cursorType = getFrame().getDrawSpace().getCursor().getType();
    int x = cursorType == Cursor.S_RESIZE_CURSOR
        ? AbstractTool.drawWidth : e.getX();
    int y = cursorType == Cursor.W_RESIZE_CURSOR
        ? AbstractTool.drawHeight : e.getY();
    MyImage img = null;
    //如果鼠标指针是拖动状态
    if ( cursorType == Cursor.NW_RESIZE_CURSOR
        || cursorType == Cursor.W_RESIZE_CURSOR
        || cursorType == Cursor.S_RESIZE_CURSOR )
        && ( x > 0 && y > 0 ) {
        //改变图像大小
        img = new MyImage( x, y, BufferedImage.TYPE_INT_RGB );
        Graphics g = img.getGraphics();
        g.setColor( Color.WHITE );
        g.drawImage( getFrame().getBufferedImage(), 0, 0
            , AbstractTool.drawWidth, AbstractTool.drawHeight, null );
        getFrame().setBufferedImage( img );
        //设置画布的大小
        AbstractTool.drawWidth = x;
        AbstractTool.drawHeight = y;
        //设置 viewport
        ImageService.setViewport( frame.getScroll()
            , frame.getDrawSpace(), x, y );
    }
}
```

这个方法由 `AbstractTool` 的 `mouseDragged` 方法调用，也就是有鼠标拖动动作时，肯定会调用到这个方法。在改变图像大小之前，需要对鼠标的当前位置做一些判断，首先获取鼠标当前的 `x` 与 `y` 坐标，如果鼠标的类型是属于改变大小类类型，而且鼠标的位置坐标 `x` 与 `y` 都大于 0（见代码中的加粗部分），那就进入改变画布大小的代码段。改变画布大小的方法是先以鼠标当前的 `x` 与 `y` 坐标创建一张新的图片，并把这张图片的底色设置为白色，再把原来的图片用 `Graphics` 的 `drawImage` 方法画到现在的画布上。`AbstractTool` 中的 `mouseDragged` 方法是对 `Tool` 接口定义的 `mouseDragged` 的实现。



### 6.5.2 实现父类的画图方法

当鼠标选择了某种工具，例如直线工具，可以在通过在画板上拖动画出一条直线，当然也可以拖动画出矩形、椭圆形等图形。在 `AbstractTool` 中，定义了一个 `draw` 方法完成这些工作，但这只是一个空方法，具体的实现由其子类去重写，因为每种工具画出的图形都不一样，所以由其子类去重写。而这个方法由一个 `createShape` 方法去调用，在 `createShape` 中，主要是把 `draw` 方法中一些共同点以同样的方式处理，避免子类的 `draw` 方法的代码重复。

代码清单：code\image\src\org\crazyit\image\tool\AbstractTool.java

```
private void createShape( MouseEvent e ,Graphics g ) {  
    //如果位置在画布内  
    if( getPressX() > 0 && getPressY() > 0  
        && e.getX() > 0 && e.getX() < AbstractTool.drawWidth  
        && e.getY() > 0 && e.getY() < AbstractTool.drawHeight ) {  
        //将整张图片重画  
        g.drawImage( getFrame().getBufferedImage(), 0, 0  
            , AbstractTool.drawWidth, AbstractTool.drawHeight, null );  
        //设置颜色  
        g.setColor( AbstractTool.color );  
        getFrame().getBufferedImage().setIsSaved( false );  
        //画图形  
        draw( g, getPressX(), getPressY(), e.getX(), e.getY() );  
    }  
}
```

如果鼠标的位置是在画布的范围内，首先将图片重画，这样做的原因是在鼠标松开之前，并没有真正把图形画出来，只是显示这个轨迹，然后再把图片的 `isSaved` 属性（是否已经保存）改变为 `false`，最后调用 `draw` 方法真正画图。我们要明白何时调用 `createShape` 的方法，当鼠标拖动或者松开的时候，就需要调用画图的方法，将图画到界面中，也就是在调用 `mouseReleased` 与 `mouseDragged` 方法的时候，就需要调用这个 `createShape` 的工具方法。

在实现这个工具方法的过程中，我们不知不觉使用了“模板方法（TemplateMethod）”这一种设计模式，模板方法是定义某个操作中的算法结构，而将这个结构中的某一步具体步骤延迟到子类中加载，这个设计模式可以使子类不必去改变整个算法的结构而重新定义该算法的某个具体步骤。在以上的代码中，我们在 `createShape` 中定义了具体画图的步骤，但是，以上的代码将 `draw` 方法留给了子类去实现。

### 6.5.3 鼠标移动时改变指针

当鼠标移动到画布边缘时，我们要在这种情况下改变鼠标的指针形状，为鼠标的移动加入监听器，由于是鼠标移动执行的方法，我们需要在实现 `Tool` 接口的 `mouseMoved` 方法的时候，改变鼠标的指针，鼠标有三种形状类型：`Cursor.NW_RESIZE_CURSOR`（往右下拖动）、`Cursor.W_RESIZE_CURSOR`（往右边拖动）、`Cursor.S_RESIZE_CURSOR`（往下拖动），改变鼠标形状的方法是用这三个类型中的一个去创建一个 `Cursor`，并调用 `drawSpace` 中的 `setCursor` 设置现在使用的鼠标类型，以下是 `AbstractTool` 对该功能的实现。

代码清单：code\image\src\org\crazyit\image\tool\AbstractTool.java

```
public void mouseMoved( MouseEvent e ) {  
    //获取鼠标现在的 x 与 y 坐标  
    int x = e.getX();  
    int y = e.getY();  
    //获取默认鼠标指针  
    Cursor cursor = getDefaultCursor();
```

```

//如果鼠标指针在右下角
if( x > AbstractTool.drawWidth - 4
    && x < AbstractTool.drawWidth + 4
    && y > AbstractTool.drawHeight - 4
    && y < AbstractTool.drawHeight + 4 ) {
    //将鼠标指针改变为右下拖动形状
    cursor = new Cursor( Cursor.NW_RESIZE_CURSOR );
}
//如果鼠标指针在右中
if( x > AbstractTool.drawWidth - 4
    && x < AbstractTool.drawWidth + 4
    && y > (int)AbstractTool.drawHeight/2 - 4
    && y < (int)AbstractTool.drawHeight/2 + 4 ){
    //将鼠标指针改变为右拖动形状
    cursor = new Cursor( Cursor.W_RESIZE_CURSOR );
}
//如果鼠标指针在下中
if( y > AbstractTool.drawHeight - 4
    && y < AbstractTool.drawHeight + 4
    && x > (int)AbstractTool.drawWidth/2 - 4
    && x < (int)AbstractTool.drawWidth/2 + 4 ) {
    //将鼠标指针改变为下拖动形状
    cursor = new Cursor( Cursor.S_RESIZE_CURSOR );
}
//设置鼠标指针类型
getFrame().getDrawSpace().setCursor( cursor );
}

```

#### 6.5.4 记录记录鼠标按下的位置

在使用工具时，不管是画直线或者矩形，总是需要记录鼠标按下时的坐标位置，加上鼠标松开时的坐标位置，才能把一个图形准确地画出来，所以在 `mousePressed` 方法中，主要是记录鼠标按下时的位置，以下是 `AbstractTool` 的 `mousePressed` 方法的实现。

代码清单：`code\image\src\org\crazyit\image\tool\AbstractTool.java`

```

public void mousePressed(MouseEvent e) {
    //如果位置在图片范围内，设置按下的坐标
    if( e.getX() > 0 && e.getX() < AbstractTool.drawWidth
        && e.getY() > 0 && e.getY() < AbstractTool.drawHeight ) {
        setPressX( e.getX() );
        setPressY( e.getY() );
    }
}

```

#### 6.5.5 重绘图片

每次松开鼠标时，可能就是一个画图动作的完成，所以 `AbstractTool` 在实现 `Tool` 的 `mouseReleased` 方法时需要做两步工作，一个是再次调用 `createShape` 方法去画图，另外一个则是调用 `drawSpace` 的 `repaint` 方法去刷新，见以下代码。

代码清单: code\image\src\org\crazyit\image\tool\AbstractTool.java

```
public void mouseReleased( MouseEvent e ) {  
    //画图  
    Graphics g = getFrame().getBufferedImage().getGraphics();  
    createShape( e, g );  
    //把 pressX 与 pressY 设置为初始值  
    setPressX( -1 );  
    setPressY( -1 );  
    //重绘  
    getFrame().getDrawSpace().repaint();  
}
```

到此, **AbstractTool** 都已经实现, **AbstractTool** 实现了 **Tool** 接口的全部方法, 并提供了一些方法由它的子类去实现, 接下面, 将实现各个工具的功能。

### 6.5.6 铅笔工具

铅笔工具, 实现的方法主要是在拖动鼠标的时候, 每次都画直线的形式去画下铅笔的轨迹, 先看以下代码。

代码清单: code\image\src\org\crazyit\image\tool\PencilTool.java

```
public void mouseDragged( MouseEvent e ) {  
    super.mouseDragged(e);  
    //获取图片的 Graphics 对象  
    Graphics g = getFrame().getBufferedImage().getGraphics();  
    if( getPressX() > 0 && getPressY() > 0 ) {  
        g.setColor( AbstractTool.color );  
        g.drawLine( getPressX(), getPressY(), e.getX(), e.getY() );  
        setPressX( e.getX() );  
        setPressY( e.getY() );  
        getFrame().getDrawSpace().repaint();  
    }  
}
```

首先是用 **BufferedImage** 中的 **getGraphics** 获取图片的 **Graphics** 对象 **g**, 由于我们在 **ImageFrame** 中保存到一个 **BufferedImage** 的对象 (**BufferImage** 的子类), 因此可以直接获得。如果鼠标的位置是在画布之中 (**getPressX() > 0 && getPressY() > 0**), 便把 **g** 的 **color** 设置为当前工具的颜色, 以按下时的坐标与鼠标当前的坐标位置为参数去调用 **Graphics** 的 **drawLine** 方法画直线, 并把按下时的坐标位置设置位当前坐标位置, 这样做的原因是想达到铅笔的效果, 也就是说每次画直线的起点坐标都是上次的终点坐标, 最后调用 **drawSpace** 的 **repaint** 方法重绘图片。在本章中, 铅笔工具对应的是 **PencilTool** 类, 使用铅笔工具效果如图 6.8 所示。

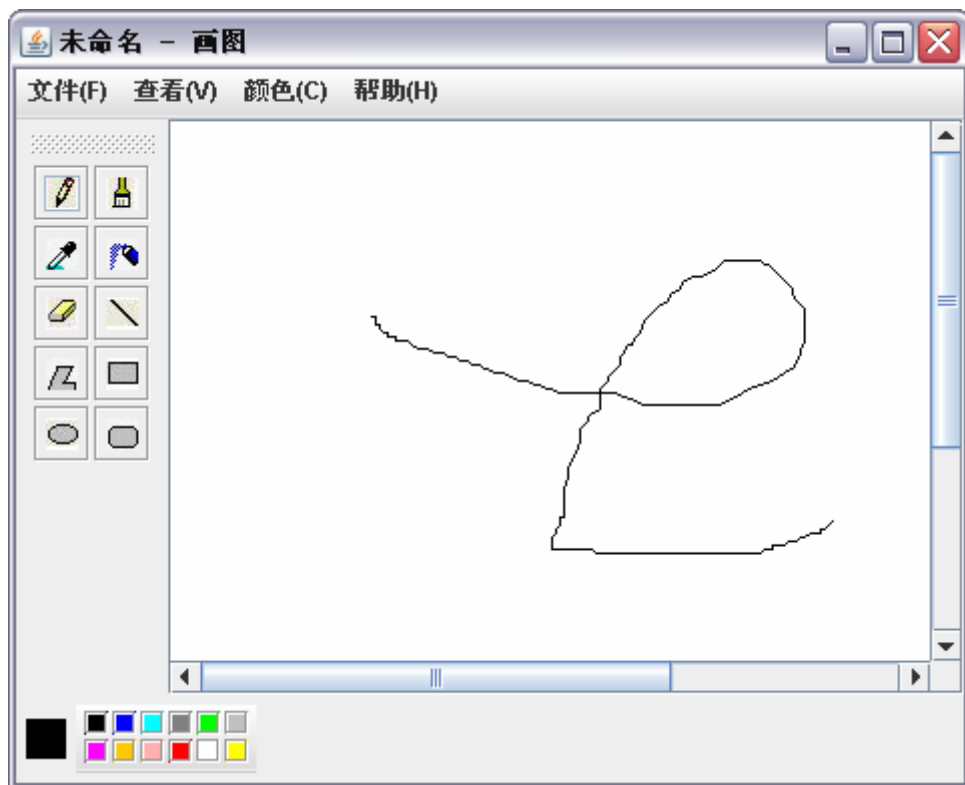


图 6.8 使用铅笔工具画图

### 6.5.7 直线工具、矩形工具、椭圆工具和圆矩形工具

这几个类分别代表直线工具、矩形工具、椭圆工具、圆矩形工具，实现的方法很简单，就是调用 Graphics 的 drawXXXX 方法（drawLine、drawRect、drawOval、drawRoundRect），都重写父类的 draw（画图形方法）方法，draw 方法最后由 createShape 方法调用。

直线工具的实现如下。代码清单：code\image\src\org\crazyit\image\tool\LineTool.java

```
public void draw(Graphics g, int x1, int y1, int x2, int y2) {
    g.drawLine(x1, y1, x2, y2);
}
```

矩形工具的实现如下。代码清单：code\image\src\org\crazyit\image\tool\RectTool.java

```
public void draw(Graphics g, int x1, int y1, int x2, int y2) {
    // 计算起点
    int x = x2 > x1 ? x1 : x2;
    int y = y2 > y1 ? y1 : y2;
    // 画矩形
    g.drawRect(x, y, Math.abs(x1 - x2), Math.abs(y1 - y2));
}
```

椭圆工具的实现如下。代码清单：code\image\src\org\crazyit\image\tool\RoundTool.java

```
public void draw(Graphics g, int x1, int y1, int x2, int y2) {
    // 计算起点
    int x = x2 > x1 ? x1 : x2;
    int y = y2 > y1 ? y1 : y2;
    // 画椭圆
    g.drawOval(x, y, Math.abs(x1 - x2), Math.abs(y1 - y2));
}
```

```
}
```

圆矩形工具的实现如下。代码清单：`code\image\src\org\crazyit\image\tool\RoundRectTool.java`

```
public void draw(Graphics g, int x1, int y1, int x2, int y2) {  
    // 计算起点  
    int x = x2 > x1 ? x1 : x2;  
    int y = y2 > y1 ? y1 : y2;  
    // 画圆矩形  
    g.drawRoundRect(x, y, Math.abs(x1 - x2), Math.abs(y1 - y2), 20, 20);  
}
```

注：除了画直线外，其他的图形都需要计算起点，取较小的坐标作为图形的开始坐标。

在本章中，直线工具对应的是 **LineTool** 类，矩形工具对应的是 **RectTool** 类，椭圆形工具对应的是，**RoundTool** 类，圆矩形工具对应的是 **RoundRectTool** 类。直线工具、矩形工具、椭圆形工具与圆矩形工具的具体效果如图 6.9 所示。

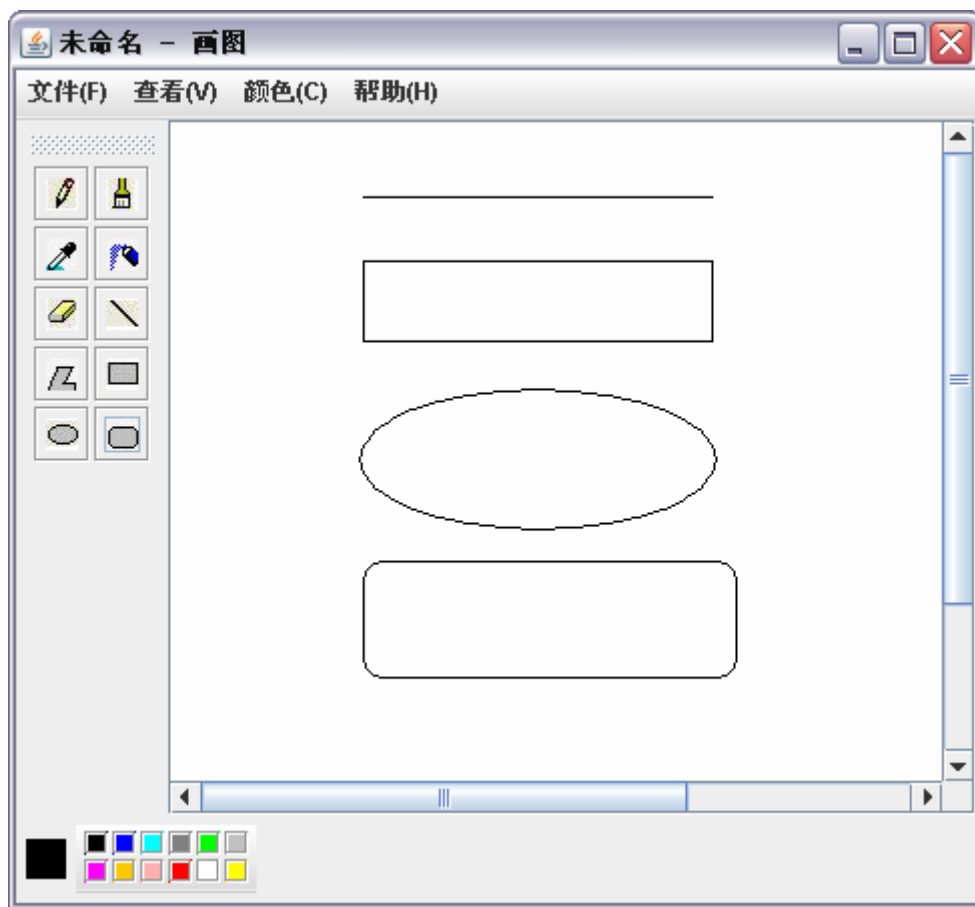


图 6.9 直线工具、矩形工具、椭圆形工具与圆矩形工具的效果

### 6.5.8 多边形工具

多边形的处理与其它图形并不一样，它不是一次性画出来的，首先是拖动画第一条边，然后松开鼠标，移动鼠标到下个位置点击松开，像画出第二条边，如此重复，最后双击鼠标闭合这个多边形。

代码中是这样实现的，每次都记录上一条直线的终点坐标，如果没有上一条线，那么终点坐标就是鼠标当前的坐标，松开鼠标的时候(`mouseReleased`)，就用上个终点坐标与鼠标当前的坐标画一条直线，在双击鼠标的时候，以鼠标当前的坐标与第一个坐标画一条直线，再以鼠标当前的坐标与最后一个坐标

画一条直线，达到闭合的目的，首先看看 `mouseReleased` 方法。

代码清单：code\image\src\org\crazyit\image\tool\PolygonTool.java

```
public void mouseReleased( MouseEvent e ) {  
    int pressX = getPressX();  
    int pressY = getPressY();  
    //调用父方法画直线  
    super.mouseReleased( e );  
    //设置第一个与最后一个坐标  
    if( firstX == -1 ) {  
        firstX = pressX;  
        firstY = pressY;  
    }  
    lastX = e.getX();  
    lastY = e.getY();  
}
```

首先是调用父方法的 `mouseReleased` 去画直线，看粗体位置的代码，如果第一个坐标还没有被赋值，那么说明这是第一次调用此方法（松开鼠标），就把这次点击松开的位置记录为第一个位置。而上一个鼠标位置就记录为当前鼠标位置。

最后双击鼠标的时候，就以当前鼠标坐标与第一个坐标（`firstX`, `firstY`）画一条直线，再以当前鼠标坐标与最后一个坐标画一条直线（`lastX`, `lastY`），见下以代码。

代码清单：code\image\src\org\crazyit\image\tool\PolygonTool.java

```
public void mouseClicked(MouseEvent e) {  
    Graphics g = getFrame().getBufferedImage().getGraphics();  
    if( e.getClickCount() == 2 && firstX > 0 && e.getX() > 0  
        && e.getX() < AbstractTool.drawWidth && e.getY() > 0  
        && e.getY() < AbstractTool.drawHeight ) {  
        g.setColor( AbstractTool.color );  
        g.drawImage( getFrame().getBufferedImage(), 0, 0  
            , AbstractTool.drawWidth, AbstractTool.drawHeight, null );  
        draw( g, 0, 0, firstX, firstY );  
        draw( g, 0, 0, lastX, lastY );  
        setPressX( -1 );  
        setPressY( -1 );  
        firstX = -1;  
        firstY = -1;  
        lastX = -1;  
        lastY = -1;  
        getFrame().getDrawSpace().repaint();  
    }  
}
```

在本章中，多边形工具对应的是 `PolygonTool` 类，多边形工具的具体效果如图 6.10 所示。

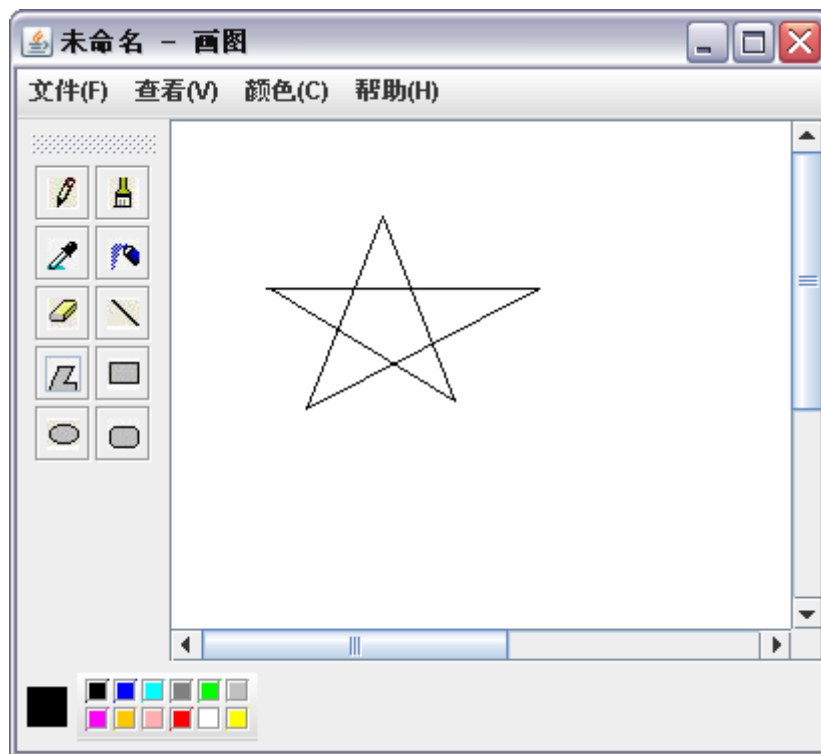


图 6.10 多边形工具

### 6.5.9 刷子与橡皮擦

刷子的效果实现：在拖动鼠标的时候，总是以当前鼠标的位置画有颜色填充的矩形，矩形的大小就是刷子的大小。而橡皮擦和刷子的实现是一样的，只不过，刷子用的是其它颜色，而橡皮擦用的是白色。见以下代码（刷子）。

代码清单：code\image\src\org\crazyit\image\tool\BrushTool.java

```
public void mouseDragged( MouseEvent e ) {
    super.mouseDragged(e);
    Graphics g = getFrame().getBufferedImage().getGraphics();
    int x = 0;
    int y = 0;
    //画笔大小
    int size = 4;
    if( getPressX() > 0 && getPressY() > 0 && e.getX() < AbstractTool.drawWidth
        && e.getY() < AbstractTool.drawHeight ) {
        g.setColor( AbstractTool.color );
        x = e.getX() - getPressX() > 0 ? getPressX() : e.getX();
        y = e.getY() - getPressY() > 0 ? getPressY() : e.getY();
        g.fillRect( x - size , y - size ,
            Math.abs( e.getX() - getPressX() ) + size
            , Math.abs( e.getY() - getPressY() ) + size );
        setPressX( e.getX() );
        setPressY( e.getY() );
        getFrame().getDrawSpace().repaint();
    }
}
```

```
}
```

在拖动鼠标过程中，总是会连续调用到这个方法，如果鼠标的当前位置是在画板中，那么就画矩形（黑体代码断是画矩形）。在本章中，刷子工具对应的是 **BrushTool** 类，橡皮擦工具对应的是 **EraserTool** 类。

### 6.5.10 喷墨工具

喷墨效果实现：点击松开鼠标的时候，以鼠标当前的位置，在喷枪大小范围内随机画出 **N** 个小矩形，见以下代码。

代码清单：code\image\src\org\crazyit\image\tool\AtomizerTool.java

```
public void draw(MouseEvent e,Graphics g) {
    int x = 0;
    int y = 0;
    //喷枪大小
    int size = 8;
    //喷枪点数
    int count = 10;
    if( getPressX() > 0 && getPressY() > 0
        && e.getX() < AbstractTool.drawWidth
        && e.getY() < AbstractTool.drawHeight ) {
        g.setColor( AbstractTool.color );
        for ( int i = 0 ; i < count ; i++ ) {
            x = new Random().nextInt(size)+ 1;
            y = new Random().nextInt(size) + 1;
            g.fillRect( e.getX() + x , e.getY() + y , 1, 1 );
        }
        setPressX( e.getX() );
        setPressY( e.getY() );
        getFrame().getDrawSpace().repaint();
    }
}
```

看加粗的代码，假如喷墨工具要每次产生 10 个小点，那么，每次都是产生两个随机数,代表与当前坐标 (x, y) 的距离 (**x = new Random().nextInt(size)+ 1; y = new Random().nextInt(size) + 1;**)，最后当前坐标加上这两个随机数，做为新的坐标去画小矩形。在本章中，喷墨工具对应的是 **AtomizerTool** 类，喷墨工具的效果如图 6.11 所示。



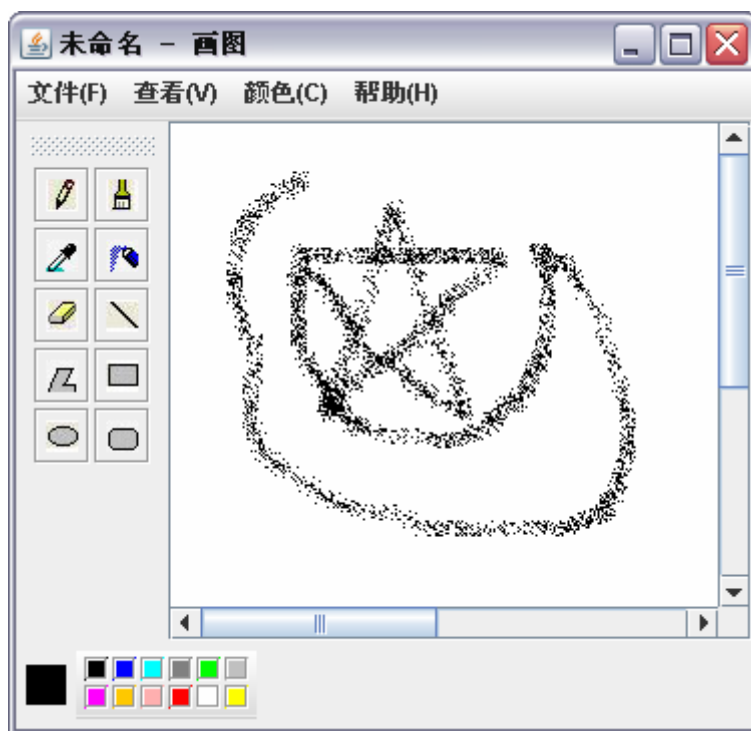


图 6.11 喷墨工具

本小节中实现画图的基本工具，包括铅笔、刷子、橡皮擦、多边形工具等画图软件的基本工具，在最后一节，将会讲解如何对图形进行保存、打开、新建等相关操作。

## 6.6 ImageService类实现

除去鼠标的画图功能，初始化画板、图片的新建打开与保存、各种面板的显示与隐藏、颜色的编辑、整个界面的刷新、菜单等业务逻辑都放在这个方法中实现。

### 6.6.1 打开图片文件

JFileChooser 用于生成打开一个文件对话框，也是用于生成“打开文件”、“保存文件”的对话框，JfileChooser 不依赖本地的 GUI 平台，它是纯 Java 实现，在不同平台具有完全的行为，并具有相同的外观风格。使用 JfileChooser 的 showOpenDialog 方法弹出文件选择框，再使用 getSelectedFile 方法获取选择到的文件。然后使用 ImageIO 的 read 方法，把文件读取出来，最后把当前的图片替换为新读取到的图片与调用 drawSpace 的 repaint 方法重画、设置窗口标题，见以下代码。

代码清单：code\image\src\org\crazyit\image\ImageService.java

```
public void open( ImageFrame frame ) {  
    save( false, frame );  
    //如果打开一个文件  
    if( fileChooser.showOpenDialog( frame )  
        == ImageFileChooser.APPROVE_OPTION ) {  
        //获取选择的文件  
        File file = fileChooser.getSelectedFile();  
        //设置当前文件夹
```

```

fileChooser.setCurrentDirectory( file );
BufferedImage image = null;
try {
    //从文件读取图片
    image = ImageIO.read( file );
} catch ( java.io.IOException e )    {
    e.printStackTrace();
}
//宽, 高
int width = image.getWidth();
int height = image.getHeight();
AbstractTool.drawWidth = width;
AbstractTool.drawHeight = height;
//创建一个 MyImage
MyImage myImage = new MyImage( width, height
    , BufferedImage.TYPE_INT_RGB );
//把读取到的图片画到 myImage 上面
myImage.getGraphics().drawImage( image, 0, 0, width
    , height, null );
frame.setBufferedImage( myImage );
//repaint
frame.getDrawSpace().repaint();
//重新设置 viewport
ImageService.setViewport( frame.getScroll()
    , frame.getDrawSpace(), width, height );
//设置保存后的窗口标题
frame.setTitle( fileChooser.getSelectedFile().getName()
    + " - 画图" );
}
}

```

### 6.6.2 保存图片

保存图片的操作一样是使用 `JFileChooser`，首先是用 `JFileChooser` 的 `getCurrentDirectory()` 方法获取当前的文件目录，以这个当前路径加上文件名做为文件保存的路径。最后，使用 `ImageIO` 的 `write` 方法把文件写到磁盘。

在这里还有另外一个逻辑，就是看 `save` 方法传过来的 `boolean` 类型的参数 `b`，如果为真，直接保存，如果为假，先询问是否保存，如果用户选择是，再以 `true` 去调用这个方法保存文件，见以下代码。

代码清单：`code\image\src\org\crazyit\image\ImageService.java`

```

public void save( boolean b, ImageFrame frame ) {
    if( b ) {
        //如果选择保存
        if( fileChooser.showSaveDialog( frame )
            == ImageFileChooser.APPROVE_OPTION ) {
            //获取当前路径
            File currentDirectory = fileChooser.getCurrentDirectory();
            //获取文件名
            String fileName = fileChooser.getSelectedFile().getName();
            //获取后缀名

```

```

        String suf = fileChooser.getSuf();
        //组合保存路径
        String savePath = currentDirectory + "\\"
            + fileName + "." + suf;
        try {
            //将图片写到保存路径
            ImageIO.write( frame.getBufferedImage(), suf
                , new File(savePath) );
        } catch ( java.io.IOException ie ) {
            ie.printStackTrace();
        }
        //设置保存后的窗口标题
        frame.setTitle( fileName + "." + suf + " - 画图" );
        //已保存
        frame.getBufferedImage().setIsSaved( true );
    }
}
else if( !frame.getBufferedImage().isSaved() ) {
    //新建一个对话框
    JOptionPane option = new JOptionPane();
    //显示确认保存对话框 YES_NO_OPTION
    int checked = option.showConfirmDialog( frame, "保存改动?"
        , "画图", option.YES_NO_OPTION, option.WARNING_MESSAGE);
    //如果选择是
    if( checked == option.YES_OPTION ) {
        //保存图片
        save( true, frame );
    }
}
}
}

```

### 6.6.3 新建图片

调用新建图片方法的时候，首先会先调用 **save** 方法，询问是否保存当前正在编辑的图片，先做用户选择是否保存的动作。然后又重新创建一个图片对象，设置这个图片的长、宽，设置图片的颜色为白色，把它替换到当前剪辑的图片，最后再设置窗口标题等其它东西，见以下代码：

代码清单：code\image\src\org\crazyit\image\ImageService.java

```

public void createGraphics( ImageFrame frame ) {
    save( false, frame );
    //宽，高
    int width = (int)getScreenSize().getWidth()/2;
    int height = (int)getScreenSize().getHeight()/2;
    AbstractTool.drawWidth = width;
    AbstractTool.drawHeight = height;
    //创建一个 MyImage
    MyImage myImage = new MyImage( width, height
        , BufferedImage.TYPE_INT_RGB );
    Graphics g = myImage.getGraphics();
    g.setColor( Color.WHITE );
}

```

```

g.fillRect( 0, 0, width, height );
frame.setBufferedImage( myImage );
//repaint
frame.getDrawSpace().repaint();
//重新设置 viewport
ImageService.setViewport( frame.getScroll()
    , frame.getDrawSpace(), width, height );
//设置保存后的窗口标题
frame.setTitle( "未命名 - 画图" );
}

```

#### 6.6.4 颜色编辑器

使用 swing 的 JColorChooser 使颜色编辑器的实现变的非常简单, 只要使用它的 showDialog 方法, 就能弹出一个颜色编辑框, 并且可以得到用户选择或者自己调的颜色, 获取到这个颜色后, 我们就把当前使用的颜色设置为这个颜色, 见以下代码。

代码清单: code\image\src\org\crazyit\image\ImageService.java

```

public void editColor( ImageFrame frame ) {
    //获取颜色
    Color color = JColorChooser
        .showDialog( frame.getColorChooser()
            , "编辑颜色", Color.BLACK );
    color = color == null ? AbstractTool.color : color;
    //设置工具的颜色
    AbstractTool.color = color;
    //设置目前显示的颜色
    frame.getCurrentColorPanel().setBackground( color );
}

```

粗体的地方, 是弹出一个标题为“编辑颜色”, 默认选择了黑色的颜色编辑框, 在用户未确定之前, 代码不会继续往下走, 等用户选择了, 就返回一个 Color 对象, 然后我们使用这个对象改变当前工具的颜色。

#### 6.6.5 文件过滤

由于我们只会选择图片类型的文件, 也只会保存为图片类型的文件, 所以对 JFileChooser 选择的文件必须做一些过滤, 我们这里选择了扩展 JFileChooser 类, 也就是继承它。这里定义了它的一个子类 ImageFileChooser, 在这个类里面, 将对文件做一些过滤, 见以下代码。

代码清单: code\image\src\org\crazyit\image\ImageFileChooser.java

```

private void addFilter() {
    this.addChoosableFileFilter( new MyFileFilter(
        new String[]{ ".BMP" }, "BMP (*.BMP)" ) );
    this.addChoosableFileFilter( new MyFileFilter(
        new String[]{ ".JPG", ".JPEG", ".JPE", ".JFIF" },
        "JPEG (*.JPG;*.JPEG;*.JPE;*.JFIF)" );
    this.addChoosableFileFilter( new MyFileFilter(
        new String[]{ ".GIF" }, "GIF (*.GIF)" );
    this.addChoosableFileFilter( new MyFileFilter(
        new String[]{ ".TIF", ".TIFF" }, "TIFF (*.TIF;*.TIFF)" );
}

```

```

        this.addChoosableFileFilter( new MyFileFilter(
            new String[]{ ".PNG", "PNG (*.PNG)" } );
        this.addChoosableFileFilter( new MyFileFilter(
            new String[]{ ".ICO", "ICO (*.ICO)" } );
        this.addChoosableFileFilter( new MyFileFilter(
            new String[]{ ".BMP", ".JPG", ".JPEG", ".JPE", ".JFIF",
                ".GIF", ".TIF", ".TIFF", ".PNG", ".ICO",
                "所有图形文件" } );
    }

```

从代码中可以看到，就是调用 `JFileChooser` 的 `addChoosableFileFilter` 为文件增加各种图片类型的过滤器，而这方法用到一个 `FileFilter` 类型的参数，在这里，我们也扩展了 `FileFilter`，叫 `MyFileFilter`，是 `ImageFileChooser` 的一个内部类，并且重写了 `FileFilter` 的 `accept` 方法，`accept` 方法是用来判断文件类型是否相符的，请看以下代码。

代码清单：code\image\src\org\crazyit\image\ImageFileChooser.java

```

public boolean accept( File f ) {
    //如果文件的后缀名合法，返回 true
    for ( String s : suffarr ) {
        if ( f.getName().toUpperCase().endsWith( s ) ) {
            return true;
        }
    }
    //如果是目录，返回 true,或者返回 false
    return f.isDirectory();
}

```

如果是目录，总是返回 `true`，代表可以打开，如果是文件，就要判断文件的后缀名是否我们自定义的后缀名，如果是就返回 `true`，否则返回 `false`；

### 6.6.6 根据菜单的点击调用相应的方法

当用户点击打开、保存、颜色等等菜单的时候，程序是根据 `ActionListener` 得到的 `cmd`（用 `ActionListener` 的 `getActionCommand` 方法获得，这个值就是菜单的文字），选择调用不同的方法，见以下代码。

代码清单：code\image\src\org\crazyit\image\ImageService.java

```

public void menuDo( ImageFrame frame, String cmd ) {
    if ( cmd.equals( "编辑颜色" ) ) {
        editColor( frame );
    }
    if ( cmd.equals( "工具箱(T)" ) ) {
        setVisible( frame.getToolPanel() );
    }
    if ( cmd.equals( "颜料盒(C)" ) ) {
        setVisible( frame.getColorPanel() );
    }
    if ( cmd.equals( "新建(N)" ) ) {
        createGraphics( frame );
    }
    if ( cmd.equals( "打开(O)" ) ) {
        open( frame );
    }
}

```

```
        if( cmd.equals( "保存(S)" ) ) {  
            save( true , frame );  
        }  
        if( cmd.equals( "退出(X)" ) ) {  
            exit( frame );  
        }  
    }  
}
```

### 6.6.7 判断图片是否已经保存

这个功能的实现非常简单，我们的图片类 `MyImage`（这个类继承了 Java 的 `BufferedImage` 类）里面有一个 `isSaved` 的属性，标志图片是否已经保存。在做保存图片的动作的时候，这个类将被调用 `setIsSave` 方法把图片 `isSaved` 设置为 `true`。如果在编辑图片的时候，这里会再把 `isSaved` 设置为 `false`。例如在 `ImageService` 的 `save()` 方法里面就有这样的语句：`frame.getBufferedImage().setIsSaved( true )`，就是把图片设置为已保存。

## 6.7 本章小结

本章主要是通过画图工具的基本实现，向读者展示了使用 `Graphic` 类的 `drawLine`、`drawRect` 等 `drawXXXX` 方法，使用 `MouseMotionListener`、`MouseListener` 等监听器这些普通的类，实现了这些画图的基本功能，加深读者对这些类与方法的理解，学会更好更有技巧地去使用这些 `swing` 包中的类。本章中除了介绍一些画图的方法外，我们还使用了模板方法这个设计模式，并将工具抽象成一个接口，使得我们编写的程序耦合性更低，向读者展现了面向对象的多态、继承与封装的特性。

## 第7章 单机连连看

### 7.1 连连看游戏简介

连连看游戏，是指在一个区域内，分布着许多不同的图片，通过点击两张相同的图片并符合一定的规则消除它们的一个游戏。我们最常见的是在 QQ 游戏大厅里的 QQ 连连看，网络上也有许多各式各样的单机版连连看游戏。在游戏的过程中，可以考虑下如何使用自己掌握的程序去实现游戏的效果，并尝试去开发属于自己的连连看，这是一件十分惬意的事情。在本章中，我们将详细的教大家如何使用 Java 去开发一款属于自己的单机连连看。

### 7.2 连连看游戏原理

实现连连看游戏并不复杂，如果先了解了程序实现这个游戏的原理，那么写出程序也就是一件简单的事情了。首先，我们可以将玩游戏区域在 Java 程序中看作是一个二维数组对象，游戏区中可以看作是一个容器对象，二维数组中一维的值可以看作是游戏区域中 x 坐标的值，二维数组中二维的值可以看作是游戏区域中 y 坐标的值，容器根据这个二维数组去构造游戏区域。如下代码创建一个二维数组：

```
Object[][] array = new Object[一维的最大值][二维的最大值];
```

其次，当点击游戏区域的某个点时，我们可以找到该点在二维数组中对应的某个值。如果之前点击的坐标点和本次点击的坐标点，它们拥有同一张图片并且符合一定的消除规则，那么，把二维数组中对应的值设置成另外的值，再重新绘制游戏区域，就可以达到消除的效果。

最后，我们可以为游戏加入分数计算和时间计算等其他功能。下面让我们开始编写我们的连连看游戏。

### 7.3 创建游戏界面与游戏区域

了解了连连看游戏的实现原理后，我们可以开始编写游戏界面，并准备游戏界面的相关类，在本章中，进行游戏的区域是 `GamePanel`，该类继承于 `JPanel`。

#### 7.3.1 创建游戏界面

游戏界面如图 7.1 所示。游戏区域中使用的是 `JPanel`。



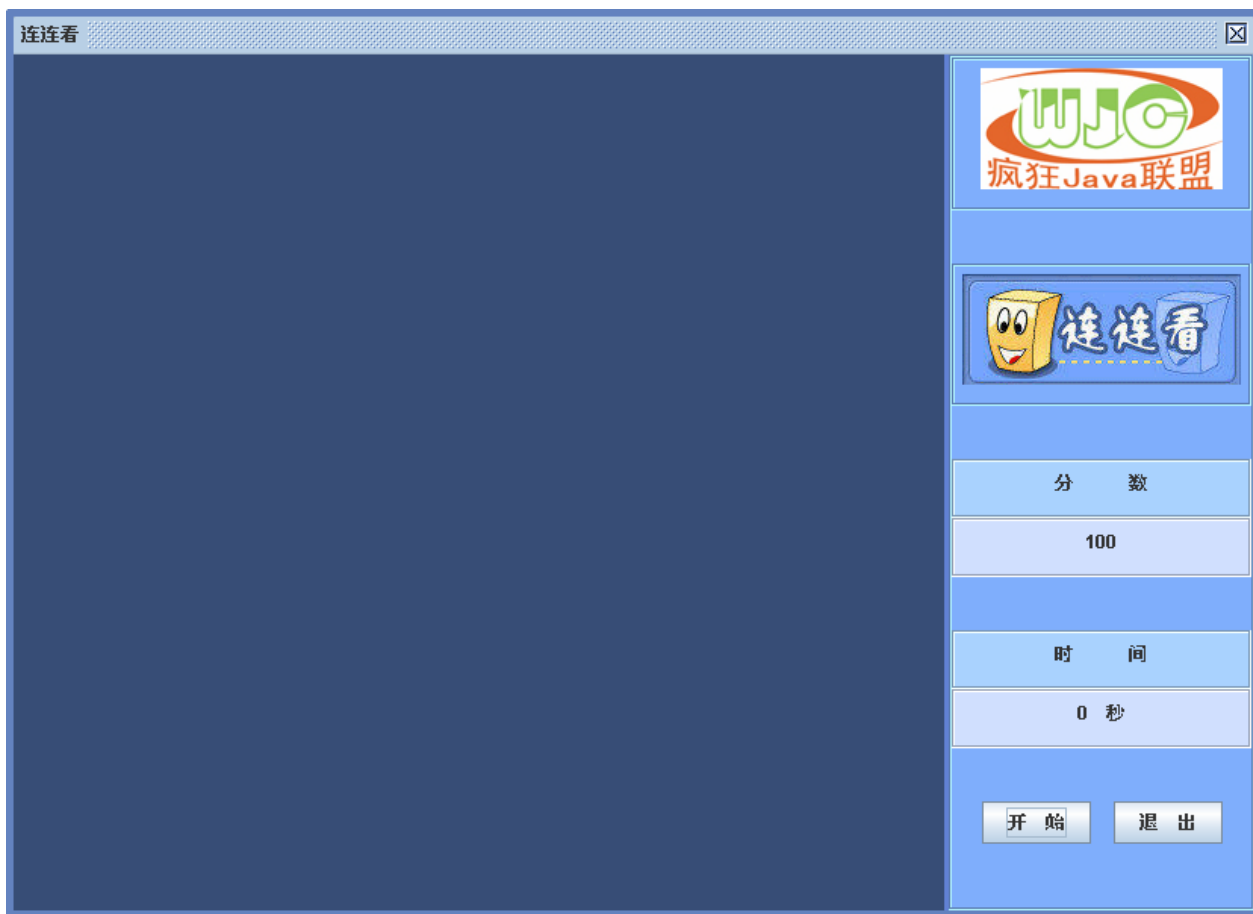


图 7.1 游戏界面

### 7.3.2 游戏区域实现原理

在本章的开头提到，我们可以将游戏区看作是一个坐标。游戏区的最左上角的那一点可以看作是坐标的  $(0, 0)$  点，游戏区的最上面的边就是  $x$  轴，游戏区最左面的边就是  $y$  轴。这样，我们就可以将游戏区域中的图片看作是一个二维数组，数组的一维值是  $x$  坐标的值，数组的二维值是  $y$  坐标的值，那么，当游戏中在游戏区域中选择了某一点的时候，我们就可以定位到该数组中的唯一的值，再去判断这个值并执行操作。

### 7.3.3 创建图片方块对象

我们首先创建一个二维数组用于表示一个游戏区域的图片，我们所需要的图片方块可以看作是一个具体的对象，因此，我们新建一个 **Piece** 类，表示为一个图片方块，这个 **Piece** 对象需要保存一些什么样的信息呢？这是我们所需要考虑的，当然，不一定一开始就把所有的属性都加入，我们可以想到的时候再往里面填充所需要的属性。一个 **Piece** 对象即一个图片方块，那么它当然会包括一个图片对象，图片占有一定的位置，那么我们需要记录它的开始的点的  $x$  坐标和  $y$  坐标，还有结束点的  $x$  坐标和  $y$  坐标，什么是 **Piece** 对象的开始坐标和结束坐标如图 7.2 所示。



图 7.2 Piece 对象的开始点与结束点

明白了什么是开始点坐标与结束点坐标之后，我们为 **Piece** 对象加入这些属性，代码如下。

代码清单：code\linkgame\src\org\crazyit\linkgame\commons\Piece.java

```
public class Piece {
    private BufferedImage image; //保存这个方块对象的所对应的图片
    private int beginX; //开始点的 x 坐标
    private int beginY; //开始点的 y 坐标
    private int endX; //结束点的 x 坐标
    private int endY; //结束点的 y 坐标
    //以下省略 getter 方法
}
```

我们为每个属性只提供 **getter** 方法，并不提供 **setter** 方法，我们可以通过构造器来构造这个对象，这个对象一旦构造完成后，它的每个属性都不允许改变，这是因为在创建游戏区域时这个方块在游戏区域中的位置已经固定，它的开始坐标与结束坐标都不再改变，更不会去改变它的图片，直到这个方块被消除。这里需要注意的是，我们这里的构造器参数并不需要提供结束点的 **x** 坐标和 **y** 坐标，这是因为结束点的 **x** 坐标和 **y** 坐标取决于开始点的坐标与图片的大小。

我们定义了一个方块对象，那么在定义一个二维数组的时候，我们可以这样：

```
Piece[][] pieces = new Piece[length][length];
```

另外，我们还要添加两个属性，用于记录这个 **Piece** 对象在数组中的位置是什么，便于我们得到该对象时，很轻易的定位到它在数组的位置：

```
private int indexX; //该对象在数组中一维的位置
private int indexY; //该对象在数组中二维的位置
```

注：以上两个属性，同样加到 Piece 对象的构造器中。

### 7.3.4 创建游戏处理类

新建一个 `GameService` 的接口，用于定义游戏逻辑的接口方法，再为这个接口新增一个实现类 `GameServiceImpl`，将 `GameService` 接口设置到 `GamePanel` 这个视图组件中，这们就可以达到逻辑与视图分离了。

代码清单：code\linkgame\src\org\crazyit\linkgame\service\GameService.java

```
public interface GameService {  
    //定义一个接口方法，用于返回一个二维数组  
    Piece[][] getPieces();  
}
```

以后我们需要实现某些游戏逻辑，可以在这个接口里面定义方法，并由 `GameServiceImpl` 去实现了。这样可以简化视图的代码，也遵循了单一职责的原则。现在我们为 `GamePanel` 类加入一个 `GameService` 的属性，并为它的构造器中加入设置 `GameService` 的代码，用于设置 `GameService`，使得在 `GamePanel` 中可以拿到 `GameService` 对象，代码如下。

代码清单：code\linkgame\src\org\crazyit\linkgame\view\GamePanel.java

```
private GameService gameService;  
public GamePanel(GameService gameService) {  
    //省略其他设值的代码  
}
```

现在，我们可以去修改 `GamePanel` 中的 `paint` 方法，代表游戏区域的二维数组 `Piece[][]` 的创建与变化，我们可以放到 `GameService` 的实现类中去，在 `GameService` 中提供一些接口方法，`GamePanel` 通过这些方法去获取二维数组，而如何去创建，设置这个二维数组里面的值，`GamePanel` 不再需要去理会这些过程。如果 `GamePanel` 类的 `paint` 方法中可以这样得到 `Piece` 数组。

代码清单：code\linkgame\src\org\crazyit\linkgame\view\GamePanel.java

```
public void paint(Graphics g) {  
    //使用 GameService 来获取 Piece[][] 二维数组  
    Piece[][] pieces = gameService.getPieces();  
}
```

### 7.3.5 图片的读取

连连看的游戏区域中分布着图片，如果要初始化游戏区域，我们必须将这些图片读取，放到我们的二维数组中。这一小节，将介绍怎样去随机读取图片，将图片的顺序打乱等。在读取图片之前，我们先要明白连连看的游戏规则，点击两张相同的图片并符合一定的规则才能消除，换言之，随便点击一张图片，必有另外一张图片等待着与它相连，也就是在游戏区域中任何一类图片的个数都是双数。如果为单数的话，游戏将不能结束了。

首先我们必须准备好游戏中所需要的图片，本例中准备了 23 张内容不同的图片用于作为图片方块。

注：这 23 张图片的长和宽必须一样，使界面好看，也符合连连看的规则。

接下来，我们去创建用于读取图片和处理图片的 `ImageUtil` 工具类，在为这个工具类添加方法前，我们必须明确这个类的作用，读取或者处理图片的一些公共方法可以放到这里，做成静态方法给外部使用。现在我们新建一个读取图片的方法，读取某个文件夹下面的符合后缀的图片，我们把方块的图片独立放到一个文件夹，再使用程序读取它们，并封装成一个集合返回，实现代码如下。

代码清单：code\linkgame\src\org\crazyit\linkgame\utils\ImageUtil.java

```
//用于获取某个文件夹下面的所有图片  
public static List<BufferedImage> getImages(File folder, String suffix) throws IOException {
```

```

//从目标文件夹中获取文件列表
File[] items = folder.listFiles();
//创建结果集合对象
List<BufferedImage> result = new ArrayList<BufferedImage>();
//对文件列表进行遍历
for (File file : items) {
    //如果该文件符合指定的文件后缀, 则加到结果集中
    if (file.getName().endsWith(suffix)) result.add(ImageIO.read(file));
}
return result;
}

```

注: 一般作为连连看方块的图片, 每张图片的大小(长宽)都必须一致, 这是传统的规则, 本例也是采用此规则, 因此所有方块的图片的长宽必须一致。

接下来, 我们再编写一个随机读取图片的方法, 用于在一个集合中随机读取一定数量的图片, 同样地返回集合。

代码清单: code\linkgame\src\org\crazyit\linkgame\utils\ImageUtil.java

```

//随机从 sourceImages 的集合中获取 size 张图片
public static List<BufferedImage> getRandomImages(List<BufferedImage> sourceImages, int size) {
    //创建一个随机数生成器
    Random random = new Random();
    //创建结果集合
    List<BufferedImage> result = new ArrayList<BufferedImage>();
    for (int i = 0; i < size; i++) {
        try {
            //随机获取一个数字, 包括 0, 不包括源图片集合的 size
            int index = random.nextInt(sourceImages.size());
            //从源图片集合中获取该图片对象
            BufferedImage image = sourceImages.get(index);
            //添加到结果集中
            result.add(image);
        } catch (IndexOutOfBoundsException e) {
            //当源图片集合的 size 为 0 时, 会抛出数组越界的异常, 直接返回结果集
            return result;
        }
    }
    return result;
}

```

这样, 我们就可以从文件夹中读取图片作为我们游戏的方块了, 但是, 前面说到, 在游戏中每一种图片在游戏区域中出现的次数必须为双数, 游戏才可以结束, 因此, 我们需要在读取图片的时候做一些相关处理。例如, 我们的游戏区域(有图片的区域)为 10 乘 10 大小, 即横 10 个方块, 竖 10 个方块, 那么我们将要读取 100 张图片作为方块, 由于每种图片必须为双数, 因此我们将 100 除以 2, 也就是需要随机拿 50 张图片即可, 再将这 50 张图片乘以 2, 即是我们所要的 100 张图片, 也可以保证每种图片的数量是双数。现在, 我们需要有一个方法去打乱一个集合里面的元素, 就好像这样:

❑ 源: 0, 1, 2, 3, 4, 5

❑ 结果: 2, 1, 5, 0, 4, 3

以下为程序的实现。代码清单: code\linkgame\src\org\crazyit\linkgame\utils\ImageUtil.java

```

//随机打乱 sourceImages
public static List<BufferedImage> randomImages(List<BufferedImage> sourceImages) {
    //创建一个随机数生成器

```

```

Random random = new Random();
//创建一个存放数字的集合
List<Integer> numbers = new ArrayList<Integer>();
//获取一个集合，里面是一些被打乱的数字
for (int i = 0; i < sourceImages.size(); i++) {
    //随机创建一个数字，范围是 0 到参数 sourceImage 的 size，包括 0 不包括 size
    Integer temp = random.nextInt(sourceImages.size());
    //为了确保数字没有重复，如果该数字已经在存放数字的集合中，重新再获取一次数字
    if (!numbers.contains(temp)) {
        //存放数字的集合中没有该随机数，添加集合中
        numbers.add(temp);
    } else {
        //该数字已经存在于集合中，i - 1 执行循环
        i--; continue;
    }
}
//创建一个结果集合
List<BufferedImage> result = new ArrayList<BufferedImage>();
//对源图片集合进行遍历
for (int i = 0; i < sourceImages.size(); i++) {
    //从数字集合中获取已经被打乱的索引，源图片集合获取这个索引的值
    result.add(sourceImages.get(numbers.get(i)));
}
return result;
}

```

我们在这里先完成了一个工具类用于处理图片，那么在创建游戏区域的时候，我们可以不用在 **GameService** 这个逻辑类里面实现图片读取和随机打乱的功能，只需要调用这一个工具类就可以达到同种图片为双数，并随机打乱图片的功能。

### 7.3.6 创建游戏区域图片数组

在画游戏区域之前，我们必须要为游戏区域数组进行赋值，将游戏区域看作一个二维数组，那么这个数组的每个值的变化，将会影响到游戏区域的展现，因此这个二维数组对我们的游戏尤为重要。

我们现在为 **GameService** 接口提供一个 **start** 方法，用于执行游戏开始时的一些动作，例如初始化游戏区域，重新计时，重新计分等，当玩家点击了开始时，我们就要调用这个 **start** 方法开始游戏，我们可以为开始按钮创建鼠标监听器。当然，我们这里先讲创建游戏区域，**GameServiceImpl** 代码如下。

代码清单：code\linkgame\src\org\crazyit\linkgame\service\impl\GameServiceImpl.java

```

public void start() {
    //创建一个有 100 个方块的游戏区域
    this.pieces = new Piece[10][10];
    //获取游戏图片，数量为 Piece 数组一维的长度乘以二维的长度
    List<BufferedImage> playImages = ImageUtil.getPlayImages(new File("images/pieces"), 10 * 10);
    //拿第一张图片的宽，由于之前约定每张图片的大小必须一致，只拿一张即可
    int imageWidth = playImages.get(0).getWidth();
    //拿第一张图片的高
    int imageHeight = playImages.get(0).getHeight();
    //添加一个拿图片的索引
    int index = 0;
    for (int i = 0; i < this.pieces.length; i++) {

```

```

        for (int j = 0; j < this.pieces[i].length; j++) {
            //构造一个 Piece 对象，以一维的值乘以图片的宽作为该对象的开始 x 坐标
            //以二维的值乘以图片的高作为该对象的开始 y 坐标
            Piece piece = new Piece(i * imageWidth, j * imageHeight, i, j, playImages.get(index));
            //将该对象放进数组中
            this.pieces[i][j] = piece;
            index++;
        }
    }
}

```

图 7.3 对上代码作出解析。

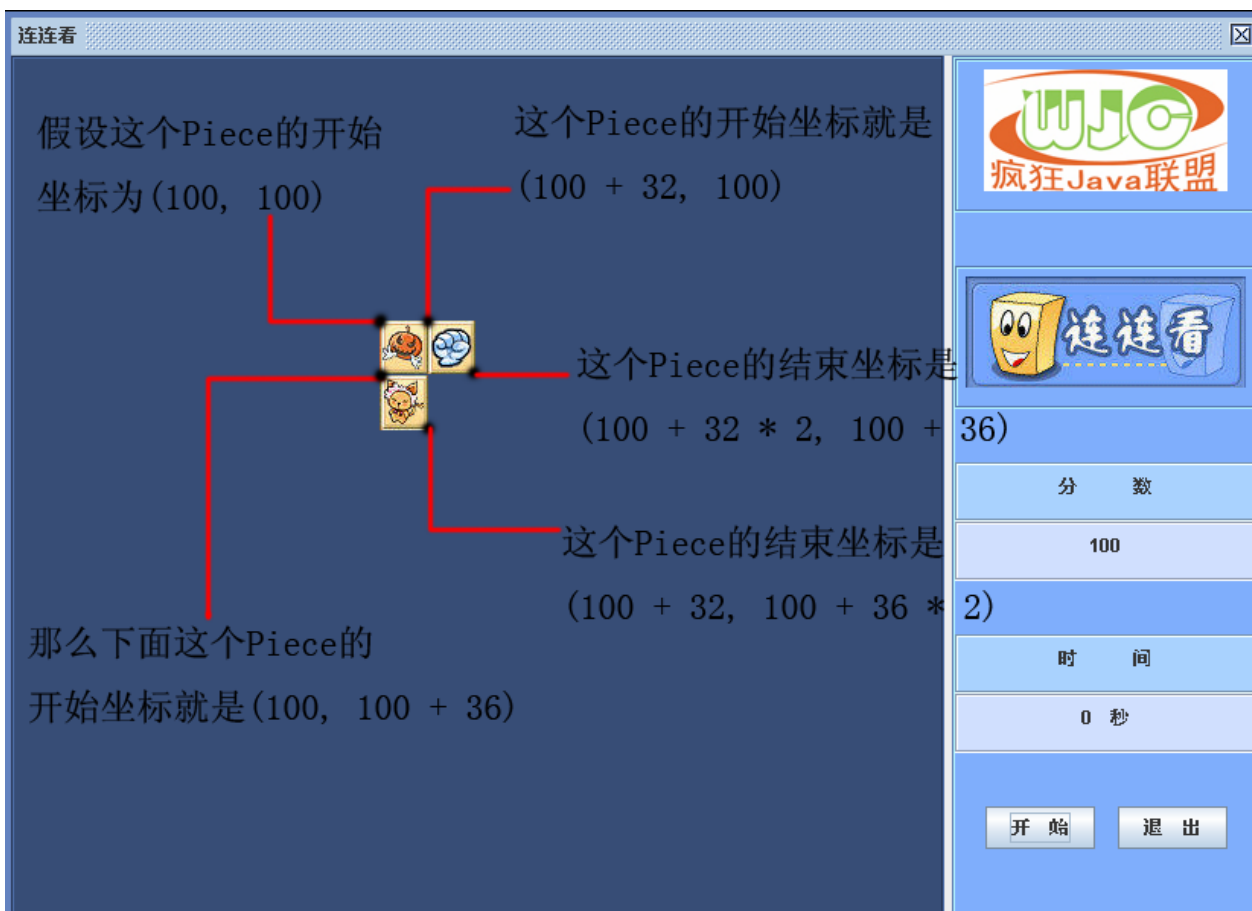


图 7.3 开始坐标与结束坐标的计算

如图 7.3，就清楚上面的代码为什么要这样实现了，现在，我们的游戏区域数组已经初始化好了，剩下的就是如果将该数组“画”到 `GamePanel` 中了。

### 7.3.7 根据数组画游戏区域

在 7.3.6 中已经创建好的游戏区域数组，现在去实现 `GamePanel` 的 `paint` 方法，用于将游戏区域数组“画”到 `GamePanel` 中，可以在对 `Piece` 数组进行循环的过程中，遇到某个非空 `Piece`，即“画”到游戏区域中，具体的代码如下。

代码清单：code\linkgame\src\org\crazyit\linkgame\view\GamePanel.java

```
//调用 Graphics 类的 drawImage 方法画图，从 piece 的开始坐标开始画
```

```
g.drawImage(piece.getImage(), piece.getBeginX(), piece.getBeginY(), null);
```

这样，就可以进这个 `GamePanel` 进行绘制了，修改 `GameServiceImpl` 类中的 `start` 方法，设置第一个方块出现的 `x` 坐标和 `y` 坐标：

```
Piece piece = new Piece(i * imageWidth + 30, j * imageHeight + 30, i, j, playImages.get(index));
```

注意上面的粗体字，代表从第一个方块开始，`x` 和 `y` 值分别加 30，即向游戏区域的右下角各移动 30。重新运行，并点击开始，效果图 7.4 所示。

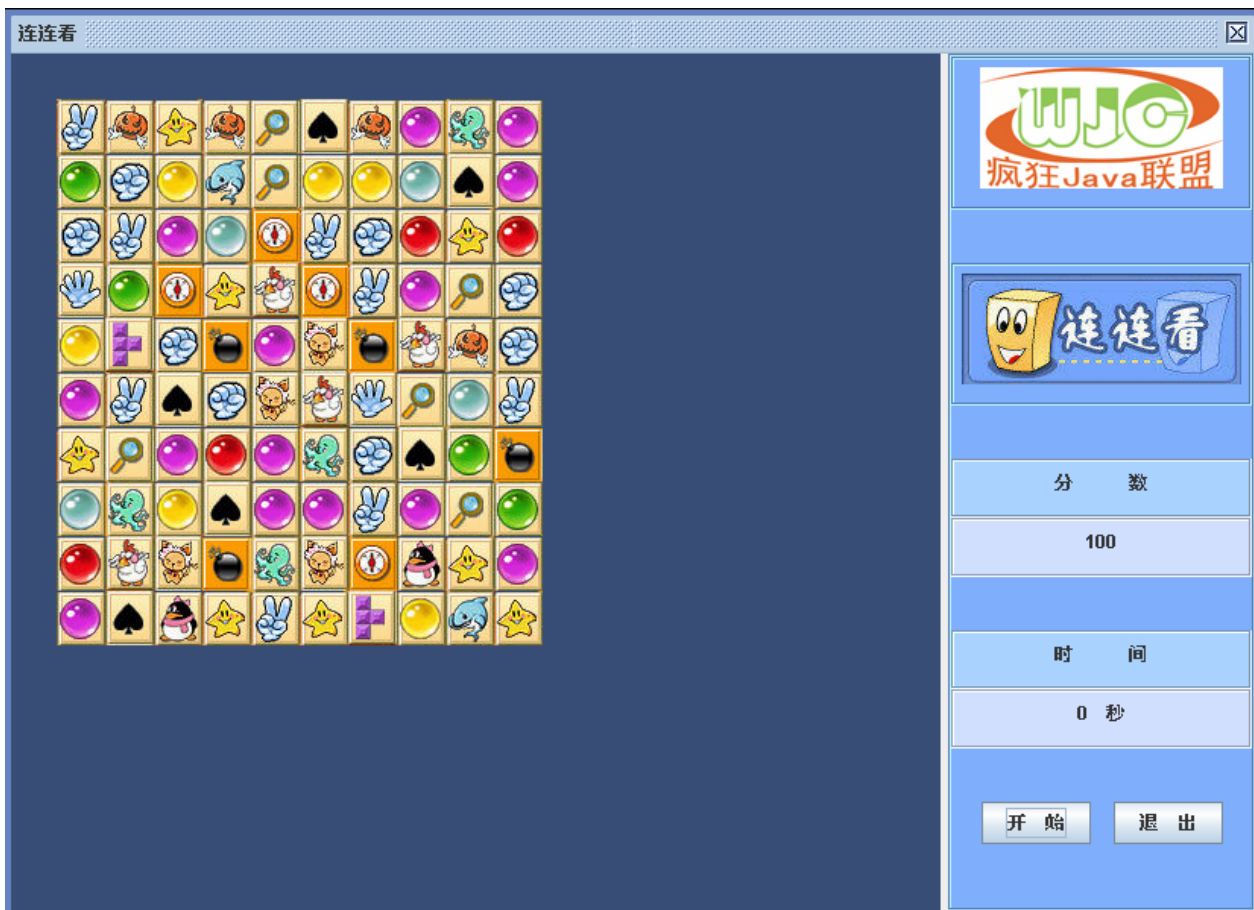


图 7.4 将二维数组“画”到游戏区

### 7.3.8 随机初始化游戏

我们定义一个 `AbstractBoard` 对象，主要用于创建游戏图片，我们可以将以上创建游戏区域的代码放置到该对象中，统一由该对象负责处理游戏图片的创建。代码如下：

代码清单：code\linkgame\src\org\crazyit\linkgame\service\AbstractBoard.java

```
//创建结果数组
Piece[][] pieces = new Piece[config.getXSize()][config.getYSize()];
//返回非空的 Piece 集合，该集合由子类去创建
List<Piece> notNullPieces = createPieces(config, pieces);
//省略以下代码
```

写一个 `SquareBoard` 类去继承 `AbstractBoard`，并实现它的 `createPieces` 方法。

代码清单：code\linkgame\src\org\crazyit\linkgame\service\impl\SquareBoard.java

```
public class SquareBoard extends AbstractBoard {
    protected List<Piece> createPieces(GameConfiguration config, Piece[][] pieces) {
```



```

List<Piece> notNullPieces = new ArrayList<Piece>();
for (int i = 0; i < pieces.length; i++) {
    for (int j = 0; j < pieces[i].length; j++) {
        //先构造一个 Piece 对象, 只设置数组中的位置为 i, j, 其他值不设置
        Piece piece = new Piece(i, j);
        notNullPieces.add(piece); //添加到非空 Piece 对象的集合中
    }
}
return notNullPieces;
}
}

```

这样, 如果还需要构造不同的游戏数组, 我们可以去继承 **AbstractBoard** 类, 并实现 **createPieces** 即可。如果现在还需要构建新的游戏数组, 那么我们再写一个子类 **SimpleBoard** 去继承 **AbstractBoard** 并实现 **createPieces** 方法, 以下是 **SimpleBoard** 的主体代码。

代码清单: code\linkgame\src\org\crazyit\linkgame\service\impl\SimpleBoard.java

```

List<Piece> notNullPieces = new ArrayList<Piece>();
for (int i = 0; i < pieces.length; i++) {
    for (int j = 0; j < pieces[i].length; j++) {
        //加入判断, 符合一定条件才去构造 Piece 对象, 并加到集合中
        if (i % 2 == 0) { //如果 x 能被 2 整除, 即单数列不会创建方块
            //先构造一个 Piece 对象, 只设置数组中的位置为 i, j, 其他值不设置
            Piece piece = new Piece(i, j);
            //添加到非空 Piece 对象的集合中
            notNullPieces.add(piece);
        }
    }
}
}

```

**createPieces** 方法中直接回返结果集合即可。现在我们编写了 **SimpleBoard** 与 **SquareBoard** 两个对象, 如果需要随机创建这两个实例, 可以通过 **Random** 来实现。游戏每次 **start** 的时候, 就会创建出不同的游戏数组了, 效果如图 7.5 所示。

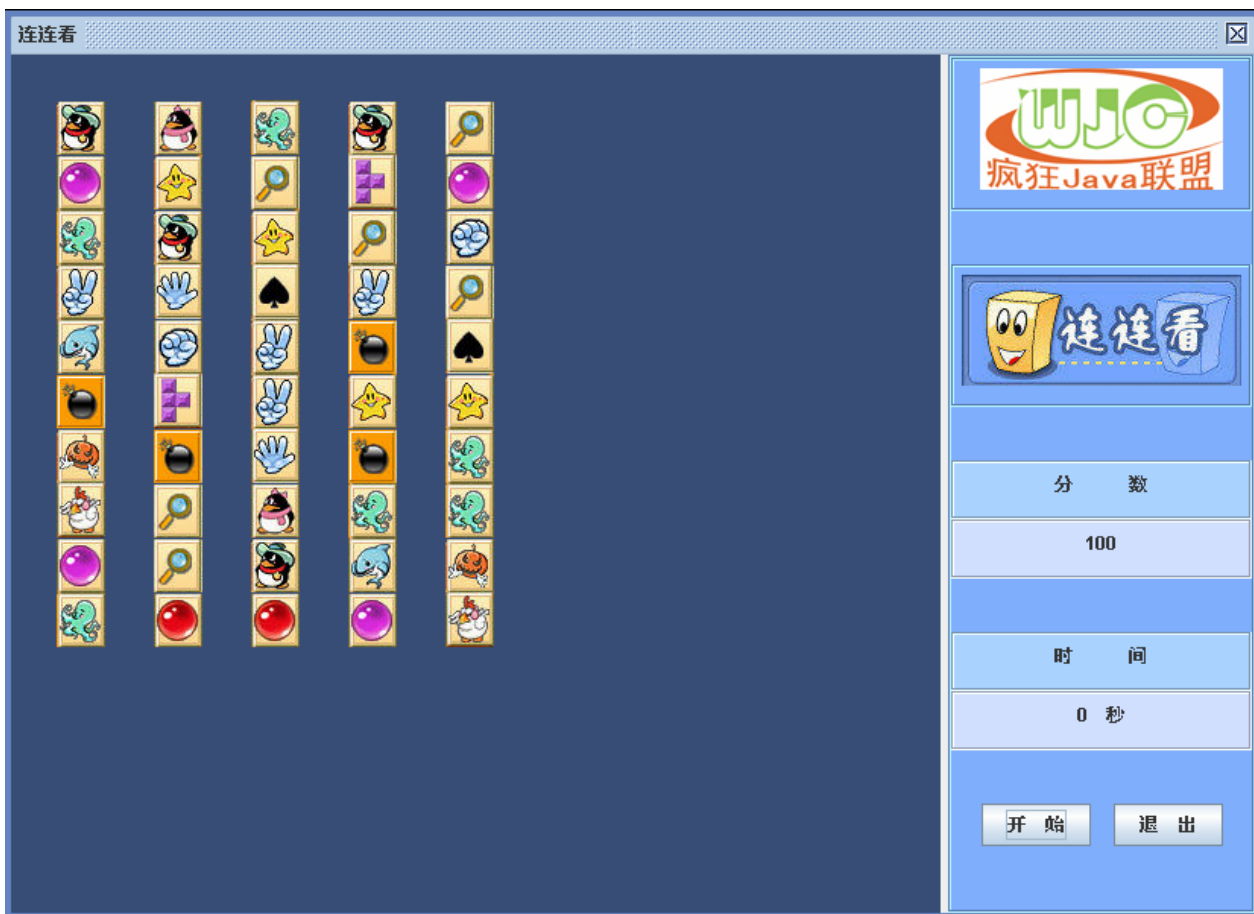


图 7.5 随机数组

其实，上面 `SquareBoard` 与 `SimpleBoard` 继承于 `AbstractBoard`，使用了其中的一种设计模式：模板方法。模板方法的概念：定义一个操作中的算法的结构，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤（摘自《设计模式》）。上面的 `SquareBoard` 与 `SimpleBoard`，继承了 `AbstractBoard`，`AbstractBoard` 中定义了图片生成的算法，它的两个子类就是实现父类定义的结构的一个步骤。

## 7.4 实现连接程序

在 7.3 中，我们已经可以创建一个随机的数组，这一节，我们开始实现连接的程序。所谓连接，就是将游戏区域内的两个相同图片按照一定的规则连接起来并消除，在这里，游戏区域中已经保存了多个 `Piece` 对象，我们在点击连接时，只需要将这些 `Piece` 对象消除就可以实现连接。

### 7.4.1 图片选择

在图片连接前，我们需要选择一张图片，并标示它已经被选择了。只需要在为鼠标的点击加入事件监听器。当鼠标在游戏区域进行点击后，根据点击的坐标得到具体的某个 `Piece` 对象。新建一个 `findPiece` 方法，根据鼠标的 `x` 坐标和 `y` 坐标（即鼠标点击的某个点）获取点击的那个 `Piece` 对象。

代码清单：code\linkgame\src\org\crazyit\linkgame\service\impl\GameServiceImpl.java

```
public Piece findPiece(int mouseX, int mouseY) {
```

```

//由于是在本类的 board 中找 Piece 对象, 如果 board 为空, 即游戏区域中没有方块
if (this.board == null) return null;
//由于我们在创建 Piece 对象的时候, 将每个 Piece 的开始坐标加了
//GameConfiguration 中设置的 beginImageX/beginImageY 值, 因此这里要减去这个值
int relativeX = mouseX - this.config.getBeginImageX();
int relativeY = mouseY - this.config.getBeginImageY();
//如果鼠标点击的地方比游戏区域中第一张图片的开始 x 坐标和开始 y 坐标要小, 即没有找到方块
if (relativeX < 0 || relativeY < 0) { return null; }
//获取 relativeX 坐标在游戏区域数组中的一维值, 第二个参数为每张图片的宽
int indexX = getIndex(relativeX, this.board.getCommonImageWidth());
//获取 relativeY 坐标在游戏区域数组中的二维值, 第二个参数为每张图片的高
int indexY = getIndex(relativeY, this.board.getCommonImageHeight());
//返回本对象中游戏区域数组的某个值
return this.pieces[indexX][indexY];
}

```

通过 `findPiece` 方法, 我们可以通过鼠标定位到数组中的某个 `Piece` 对象了, 接着, 还要处理当选择了某一个 `Piece` 对象时, 需要标识它为已经被选中的状态, 这时候需要告诉鼠标监听器, 鼠标选择了哪一个 `Piece`, 再告诉 `GamePanel` 当前选择了哪一个 `Piece`, 让 `GamePanel` 去重新绘画。这里需要注意的是 `GamePanel` 只允许出现一个被选中的 `Piece` 对象, 当选择第二个的时候, 如果可以相连, 那么将设置 `selectPiece` 为 `null`, 如果不可以相连, 那么 `selectPiece` 为选择的第二个。运行游戏并对图片进行选择效果如图 7.6 所示。

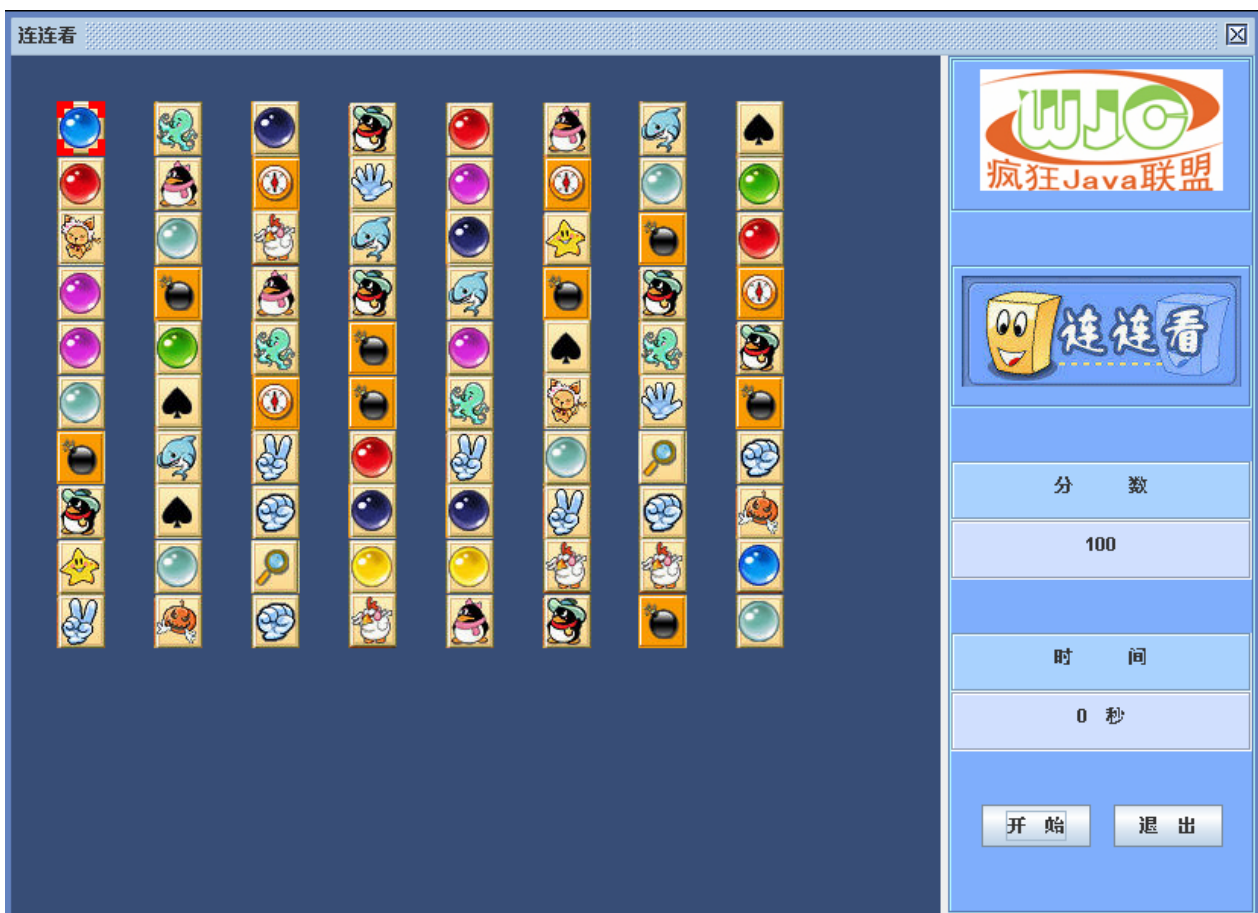


图 7.6 方块被选中的效果

图 7.6 中可以看到，游戏区域中最左上角 (0, 0) 的那个方块已经被选中。我们需要为 `GamePanel` 加入一个集合来保存已经选择的 `Piece` 对象，当选择了一个 `Piece` 对象后，将其加入到集合中。每选择一次，都需要对集合进行判断，如果集合的大小为 0，则直接加入到集合中，如果集合大小为 1，就表示之前已经选择了一个 `Piece`，就可以判断是否可以消除。

### 7.4.2 创建连接的相关对象

在实现连接功能前，我们还需要准备一些对象，用于封装在连接过程中会用到的对象，例如，连接是由某些连接点构成的，这里需要一个 `Point` 对象，连接可以看作一个 `LinkInfo` 对象，即表示连接信息。`LinkInfo` 里有一个集合属性，用于存放每一个 `Point`。具体代码如下：

代码清单：code\linkgame\src\org\crazyit\linkgame\commons\Point.java

```
public class Point {
    private int x; //记录这个点对象的 x 坐标
    private int y; //记录这个点对象的 y 坐标
    //这里需要重写 Object 的 equals 方法，用于判断两个点是否为同一个
    public boolean equals(Object object) {
        //如果 object 是 Point 类型
        if (object instanceof Point) {
            //将参数强制转成 Point 对象
            Point p = (Point)object;
            //当两个 Point 对象的 x 坐标和 y 坐标同时相等的时候，表示它们是同一个点
            return (p.getX() == this.x && p.getY() == this.y) ? true : false;
        }
        return false;
    }
    //下面省略 setter 和 getter 方法
}
```

上面新建一个 `Point` 类，用于保存点的 `x` 和 `y` 坐标，注意，该类重写的 `equals` 方法，用于判断两个 `Point` 对象是否为同一个点，判断标准为两个点的 `x` 坐标与 `y` 坐标是否一致。

代码清单：code\linkgame\src\org\crazyit\linkgame\commons\LinkInfo.java

```
public class LinkInfo {
    private List<Point> points = new ArrayList<Point>(); //创建一个集合用于保存连接点
    //返回连接集合
    public List<Point> getLinkPoints() {
        return points;
    }
}
```

该类只提供三个构造器和一个返回存放连接点集合的方法，构造器用于创造连接点信息对象，如果两点可以直接连接，则调用两个参数的构造器，两个点之间需要有一个转折点才可以相连的话，需要调用三个参数的构造器，两个点之间有两个转折点的话，就需要调用四个参数的构造器。图 7.7 解析这三个构造器的作用。

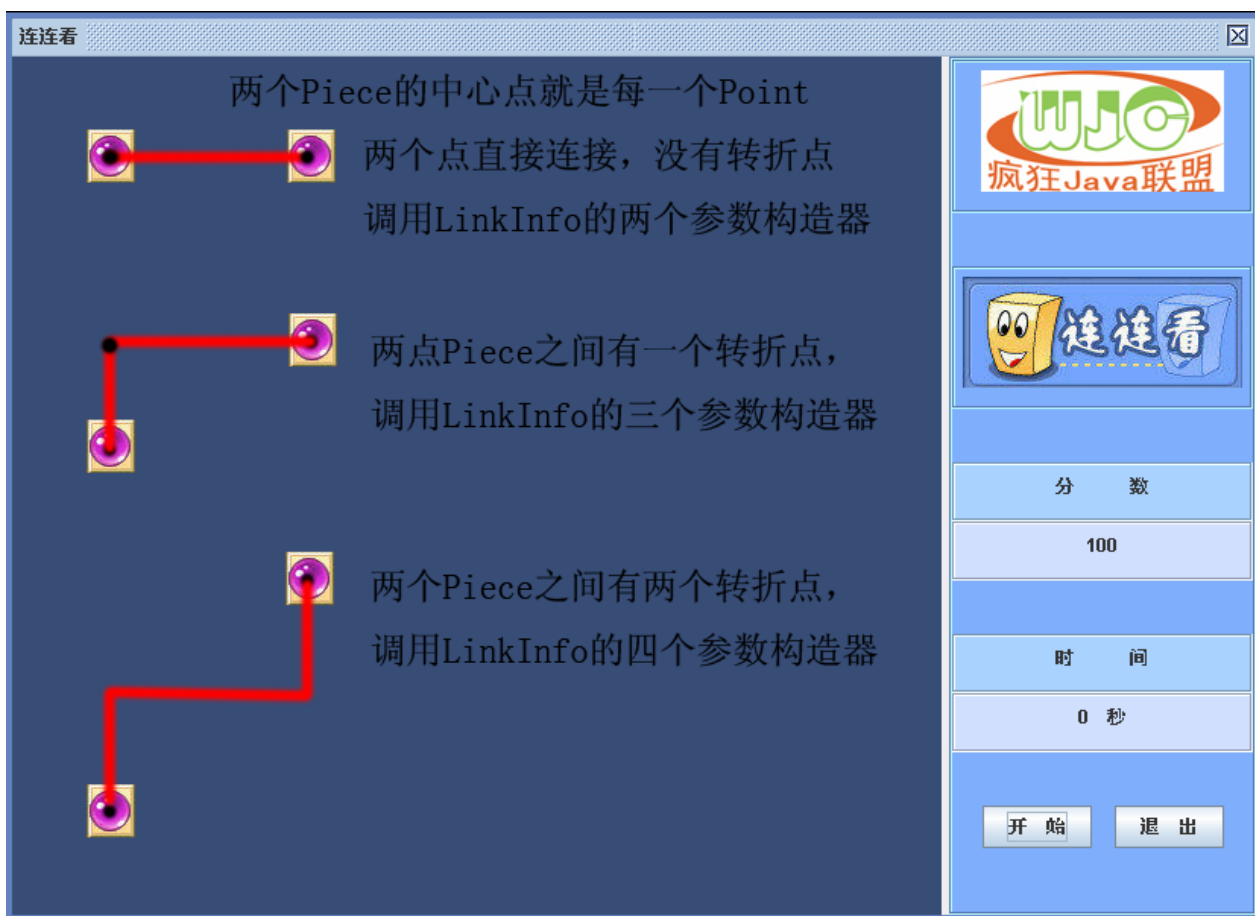


图 7.7 LinkInfo 类的描述

如图 7.7, 我下面的编码就是按照这三种方式来实现, 先判断能不能直接相连, 再判断一个转折点的情况, 最后为两个转折点的情况。我们还需要 `GameListener` 类根据 `GameService` 的 `link` 方法返回的 `LinkInfo` 对象进行处理, 让 `GamePanel` 对象进行重绘。

为 `GameService` 接口加新方法, 让 `GameListener` 调用。在 `GameService` 接口中添加 `link` 方法, 让 `GameServiceImpl` 去实现它即可, `link` 方法的参数就是两个 `Piece` 对象, 返回值就是我们上面所定义的 `LinkInfo` 对象了, 表示可否根据两个 `Piece` 对象返回一个 `LinkInfo` 对象, 如果返回 `null`, 则表示两个对象不可连接。

代码清单: `code\linkgame\src\org\crazyit\linkgame\service\GameService.java`

```
//连接两个 Piece 对象, 可以连接, 返回 LinkInfo 对象
```

```
LinkInfo link(Piece p1, Piece p2);
```

我们需要对游戏的游戏区域进行分析, 连接的情况在图 7.7 已经描述了, 那么接下来就是两个 `Piece` 在游戏区域中的关系了, 具体可分为: `p1` 与 `p2` 在同一行 (`indexY` 值相同), `p1` 与 `p2` 在同一列 (`indexX` 值相同), `p2` 在 `p1` 的右上角 (`p2` 的 `indexX` > `p1` 的 `indexX`, `p2` 的 `indexY` < `p1` 的 `indexY`), `p2` 在 `p1` 的右下角 (`p2` 的 `indexX` > `p1` 的 `indexX`, `p2` 的 `indexY` > `p1` 的 `indexY`), 这里为什么不讲 `p2` 在 `p1` 的左上角和 `p2` 在 `p1` 的左下角呢, 这种情况, 我们可以重新执行 `link` 方法, 将 `p1` 和 `p2` 两个参数的位置互换即可实现。图 7.8 讲解了几种位置的情况。

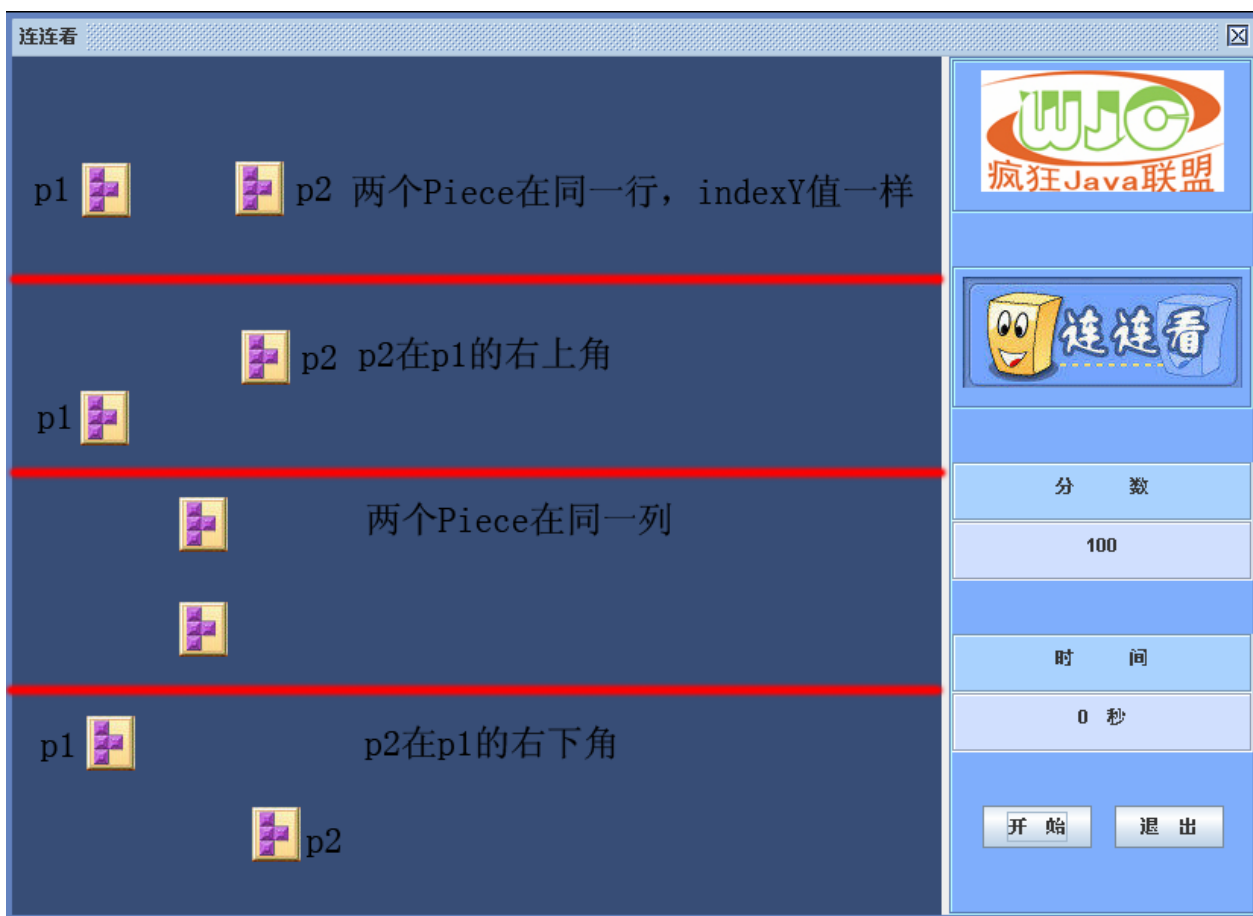


图 7.8 两个 Piece 对象的位置关系

注：如果 p2 在 p1 的左上角和左下角，也就是 p1 在 p2 的右上角和右下角，也在上面的情况中。

现在了解了这几种情况后，下面就可以着手写一些工具方法，用于处理在上面这几个情况时会用到的一些公用方法。

### 7.4.3 准备获取通道的工具方法

现在准备几个工具方法，在实现时我们需要用到的。我们需要获取一个 Piece 向上的通道，向下的通道，向左的通道，向右的通道，什么叫通道呢？如图 7.9 所示。



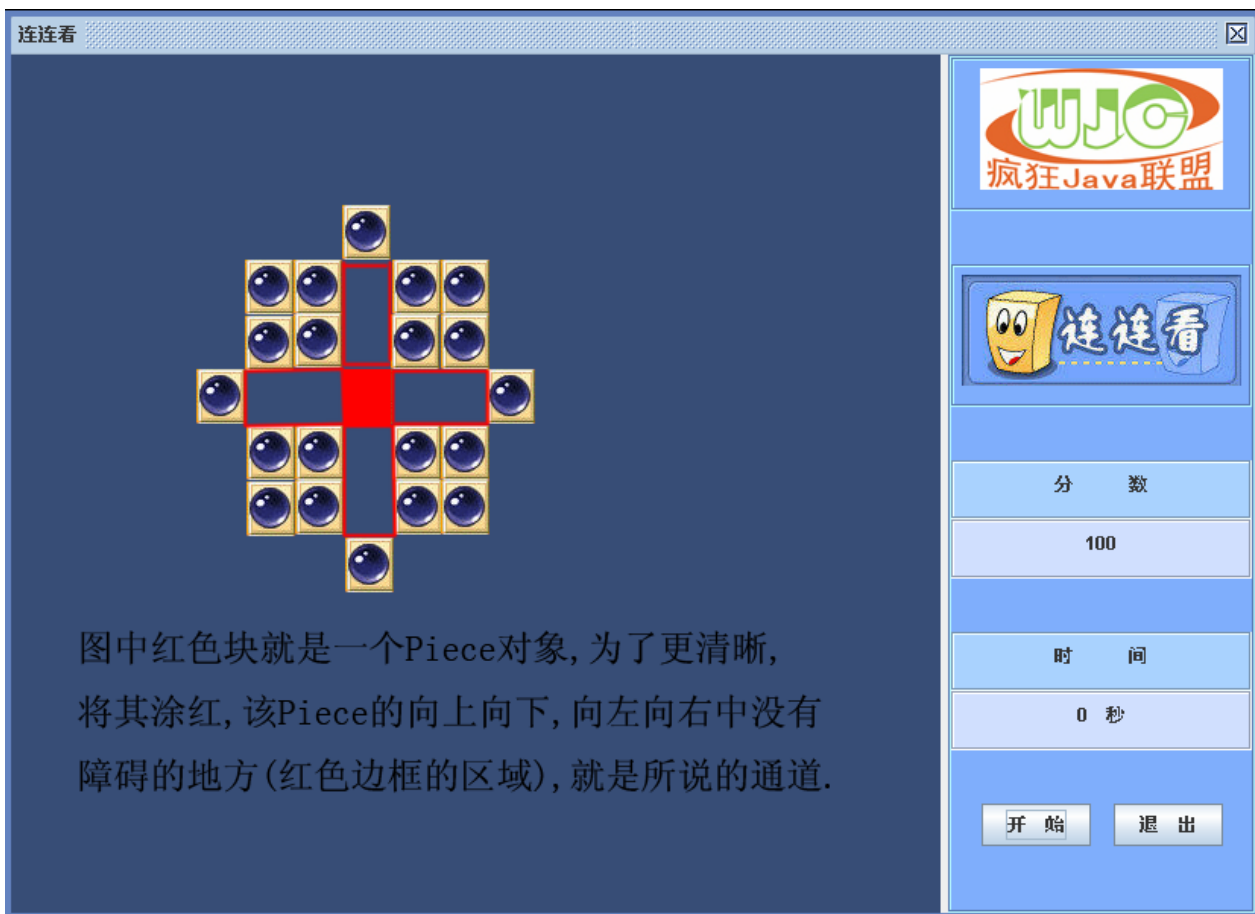


图 7.9 通道的描述

弄清楚什么是通道的概念以后, 我们写获取通道的程序:

代码清单: code\linkgame\src\org\crazyit\linkgame\service\impl\GameServiceImpl.java

```
//判断 GamePanel 中的 x, y 座标中是否有 Piece 对象
private boolean hasPiece(int x, int y) {
    return (findPiece(x, y) == null) ? false : true
}
//给一个 Point 对象,返回它的左边通道
private List<Point> getLeftChanel(Point p, int min, int pieceWidth) {
    List<Point> result = new ArrayList<Point>();
    //获取向左通道, 由一个点向左遍历, 步长为 Piece 图片的宽
    for (int i = p.getX() - pieceWidth; i >= min; i = i - pieceWidth) {
        //遇到障碍, 表示通道已经到尽头, 直接返回
        if (hasPiece(i, p.getY())) return result;
        result.add(new Point(i, p.getY()));
    }
    return result;
}
```

编写完左边通道的方法后, 按照同样的道理编写获取其他方向通道的方法。

#### 7.4.4 没有转折点的横向连接

现在让我们先实现最简单的横向连接（没有转折点），可以实现 link 方法。

代码清单：code\linkgame\src\org\crazyit\linkgame\service\impl\GameServiceImpl.java

```
//实现接口的 link 方法
public LinkInfo link(Piece p1, Piece p2) {
    //两个 Piece 是同一个，即在游戏区域中选择了同一个 Piece，返回 null
    if (p1.equals(p2)) return null;
    //如果 p1 的图片与 p2 的图片不相同，则返回 null
    if (!p1.isSameImage(p2)) return null;
}
```

上面的代码，如果在游戏区域中选择了同一个 Piece，则直接返回 null，如果 p1 与 p2 的图片不相同，表示它们不可连，直接返回。再去新增两个工具方法，用于判断一行（一列）的两个点是否有障碍。

代码清单：code\linkgame\src\org\crazyit\linkgame\service\impl\GameServiceImpl.java

```
//判断两个 y 座标相同的点对象之间是否有障碍，以 p1 为中心向右遍历
private boolean isXBlock(Point p1, Point p2, int pieceWidth) {
    //如果 p2 在 p1 左边，调换参数位置调用本方法
    if (p2.getX() < p1.getX()) return isXBlock(p2, p1, pieceWidth);
    for (int i = p1.getX() + pieceWidth; i < p2.getX(); i = i + pieceWidth) {
        //如果有障碍
        if (hasPiece(i, p1.getY())) return true;
    }
    return false;
}

//判断两个 x 座标相同的点对象之间是否有障碍，以 p1 为中心向下遍历
private boolean isYBlock(Point p1, Point p2, int pieceHeight) {
    //如果 p2 在 p1 的上面，调换参数位置重新调用本方法
    if (p2.getY() < p1.getY()) return isYBlock(p2, p1, pieceHeight);
    for (int i = p1.getY() + pieceHeight; i < p2.getY(); i = i + pieceHeight) {
        if (hasPiece(p1.getX(), i)) return true;
    }
    return false;
}
```

有了这两个方法之后，我们可以判断两个在同一行（同一列）的点之间是否可连了。下面实现同一行（同一列）可以连的情况，这里是指没有转折点的情况，修改 link 方法。

代码清单：code\linkgame\src\org\crazyit\linkgame\service\impl\GameServiceImpl.java

```
//实现接口的 link 方法
public LinkInfo link(Piece p1, Piece p2) {
    //两个 Piece 是同一个，即在游戏区域中选择了同一个 Piece，返回 null
    if (p1.equals(p2)) return null;
    //如果 p1 的图片与 p2 的图片不相同，则返回 null
    if (!p1.isSameImage(p2)) return null;
    //如果 p2 在 p1 的左边，则需要重新执行本方法，两个参数互换
    if (p2.getIndexX() < p1.getIndexX()) return link(p2, p1);
    //获取 p1 的中心点
    Point p1Point = getPieceCenter(p1);
    //获取 p2 的中心点
    Point p2Point = getPieceCenter(p2);
    //获取每张图片的宽和高
```

```

int pieceWidth = this.board.getCommonImageWidth();
int pieceHeight = this.board.getCommonImageHeight();
//如果两个 Piece 在同一行
if (p1.getIndexY() == p2.getIndexY()) {
    //它们在同一行并之间可以连，即没有直接障碍
    if (!isXBlock(p1Point, p2Point, pieceWidth)) return new LinkInfo(p1Point, p2Point);
}
return null;
}

```

现在可以测试一下，效果如图 7.10 所示。



图 7.10 实现横向相连

#### 7.4.5 没有转折点的纵向连接

下面实现纵向连接，和横向连接一样，调用工具方法 `isYBlock` 判断是否可以纵向连接即可，为 `link` 方法加入如下代码。

代码清单：code\linkgame\src\org\crazyit\linkgame\service\impl\GameServiceImpl.java

```

//如果两个 Piece 在同一列
if (p1Point.getX() == p2Point.getX()) {
    if (!isYBlock(p1Point, p2Point, pieceHeight)) { //它们之间没有直接障碍，没有转折点
        return new LinkInfo(p1Point, p2Point);
    }
}

```



```

    }
}
return null;
}

```

这样可以获取两个通道的交点，再新增一个工具方法，用于获取两个 **Point** 之间的转折点（一个 **Point**）。代码清单：`code\linkgame\src\org\crazyit\linkgame\service\impl\GameServiceImpl.java`

```

//获取两个不在同一行或者同一列的座标点的直角连接点，即只有一个转折点
private Point getCornerPoint(Point point1, Point point2, int pieceWidth, int pieceHeight) {
    //获取 p1 向右，向上，向下的三个通道
    List<Point> point1RightChanel = getRightChanel(point1, point2.getX(), pieceWidth);
    List<Point> point1UpChanel = getUpChanel(point1, point2.getY(), pieceHeight);
    List<Point> point1DownChanel = getDownChanel(point1, point2.getY(), pieceHeight);
    //获取 p2 向下，向左，向下的三个通道
    List<Point> point2DownChanel = getDownChanel(point2, point1.getY(), pieceHeight);
    List<Point> point2LeftChanel = getLeftChanel(point2, point1.getX(), pieceWidth);
    List<Point> point2UpChanel = getUpChanel(point2, point1.getY(), pieceHeight);
    if (isRightUp(point1, point2)) { //point2 在 point1 的右上角
        //获取 p1 向右和 p2 向下的交点
        Point linkPoint1 = getWrapPoint(point1RightChanel, point2DownChanel);
        //获取 p1 向上和 p2 向左的交点
        Point linkPoint2 = getWrapPoint(point1UpChanel, point2LeftChanel);
        //返回其中一个交点，如果没有交点，则返回 null
        return (linkPoint1 == null) ? linkPoint2 : linkPoint1;
    }
    if (isRightDown(point1, point2)) { //point2 在 point1 的右下角
        //获取 p1 向下和 p2 向左的交点
        Point linkPoint1 = getWrapPoint(point1DownChanel, point2LeftChanel);
        //获取 p1 向右和 p2 向下的交点
        Point linkPoint2 = getWrapPoint(point1RightChanel, point2UpChanel);
        return (linkPoint1 == null) ? linkPoint2 : linkPoint1;
    }
    return null;
}

```

以上的代码获得两个 **Piece** 之间的转折点，如图 7.12 所示。

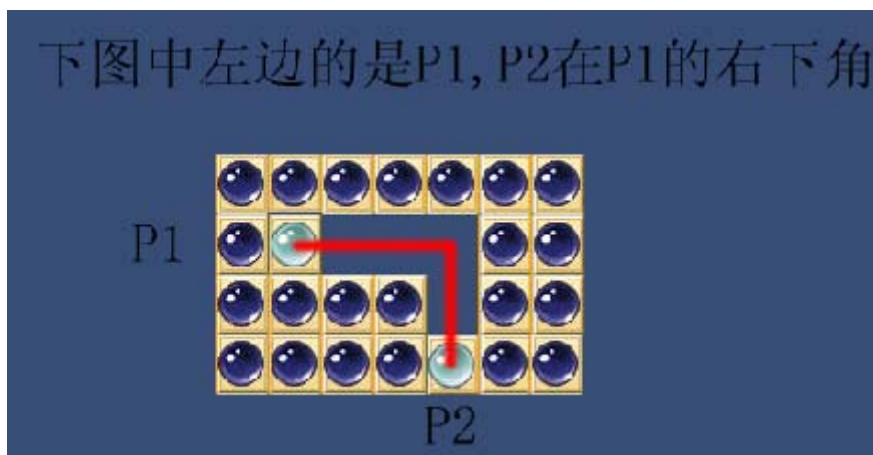


图 7.12 一个转折点，p2 在 p1 的右下角

图 7.12 的情况，我们需要获取 p1 向右的通道，再获取 p2 向上的通道，再判断两个通道间是否有交点，并返回该点即可。当 p2 在 p1 的右上角的情况如图 7.13 所示。

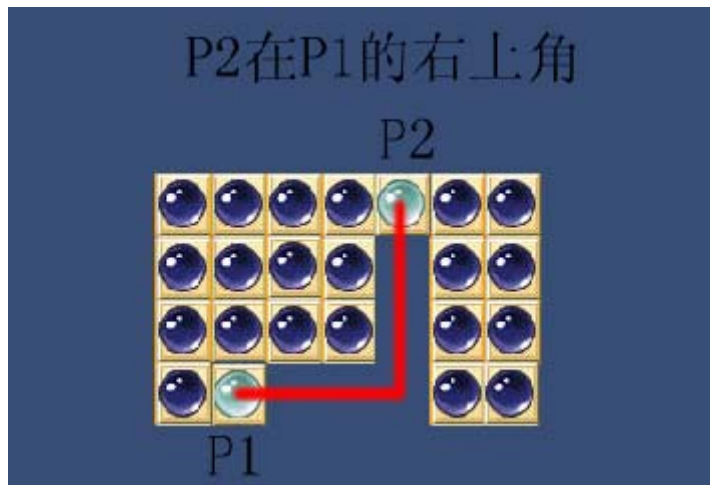


图 7.13 一个转折点，p2 在 p1 的右上角

p2 在 p1 的右上角，即获取 p1 的上右通道，p2 的向下通道，再获取它们的交点即可。在 GameServiceImpl 的 link 方法中加入判断代码，到现在 link 方法的代码如下。

代码清单：code\linkgame\src\org\crazyit\linkgame\service\impl\GameServiceImpl.java

```
//实现接口的 link 方法
public LinkInfo link(Piece p1, Piece p2) {
    //省略前面横向直接相连与纵向直接相连的代码
    //有一个转折点的情况
    //获取两个点的直角相连的点，即只有一个转折点
    Point cornerPoint = getCornerPoint(p1Point, p2Point, pieceWidth, pieceHeight);
    if (cornerPoint != null) return new LinkInfo(p1Point, cornerPoint, p2Point);
    return null;
}
```

#### 7.4.7 两个转折点的连接

上一小节讲了两个 Point 之间一个转折点的情况，这一节，将实现两个 Point 通过两个转折点进行相连，两个转折点的情况比较多，可以总结为以下几种：

- ❑ 在同一行，不能直接相连，就必须有两个转折点，分向上与向下两种连接情况。
- ❑ 在同一列，不能直接相连，也必须有两个转折点，分向左与向右两种连接情况。
- ❑ p2 在 p1 的右下角，这里就有六种转折情况。
- ❑ p2 在 p1 的左上角，同样地也有六种转折情况。

我们一种一种来解决，先解决在同一行有两个转折点的情况。请看图 7.14，说明它们是如何相连的。



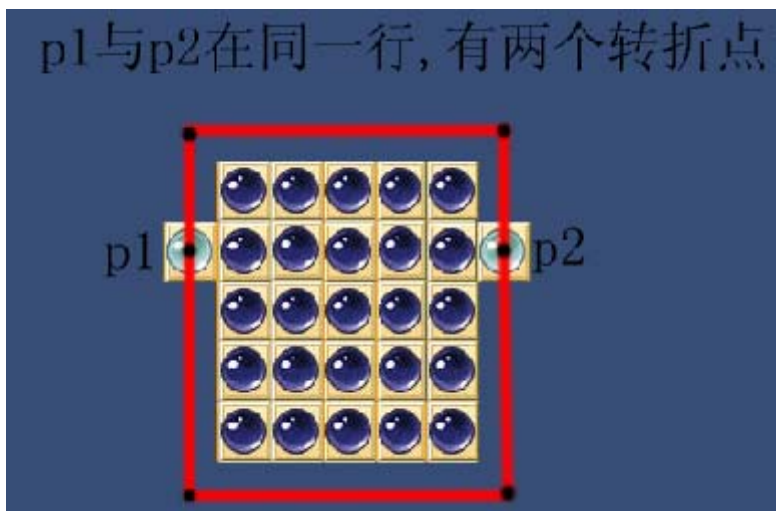


图 7.14 两个转折点, p1 与 p2 在同一行

图 7.14 可以看到, 它们 p1 与 p2 相连, 可以在上面连, 也可以在下面连, 这两种情况都代表它们可以相连, 我们先把这两种情况都加入结果中, 到最后再去计算最近的距离。实现时我们可以先构建一个 Map, Map 的 key 为第一个转折点, map 的 value 为第二个转折点, 如 map 的 size 不止 1 的话, 证明这两个 Point 有多种连接途径, 我们先返回第一个连接途径, 最后再计算最小的连接方式。为 link 方法添加如下代码。

代码清单: code\linkgame\src\org\crazyit\linkgame\service\impl\GameServiceImpl.java

```
//该 map 的 key 存放第一个转折点, value 存放第二个转折点, map 的 size 说明有多少个可以连的途径
Map<Point, Point> turns = getLinkPoints(p1Point, p2Point, pieceWidth, pieceHeight);
if (turns.size() != 0) {
    for (Object turn : turns.keySet()) {
        //遍历该 map, 暂时只返回 map 中第一个元素
        //获取 map 中第一个元素的 key 值, 即第一个转折点
        Point point1 = (Point)turn;
        //获取 map 中第一个元素的 value 值, 即第二个转折点
        Point point2 = turns.get(point1);
        return new LinkInfo(p1Point, point1, point2, p2Point);
    }
}
```

上面的代码有点难以理解, 可以看图 7.15。

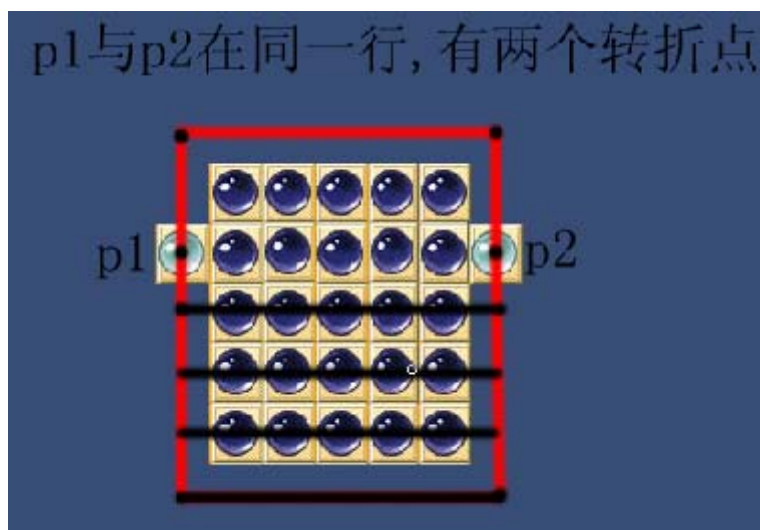


图 7.15 横向两个转折点的代码解释

图 7.15 中的几条黑线，如果这几条黑线间没有障碍，则最终会加到结果的 map 中。效果如图 7.16。

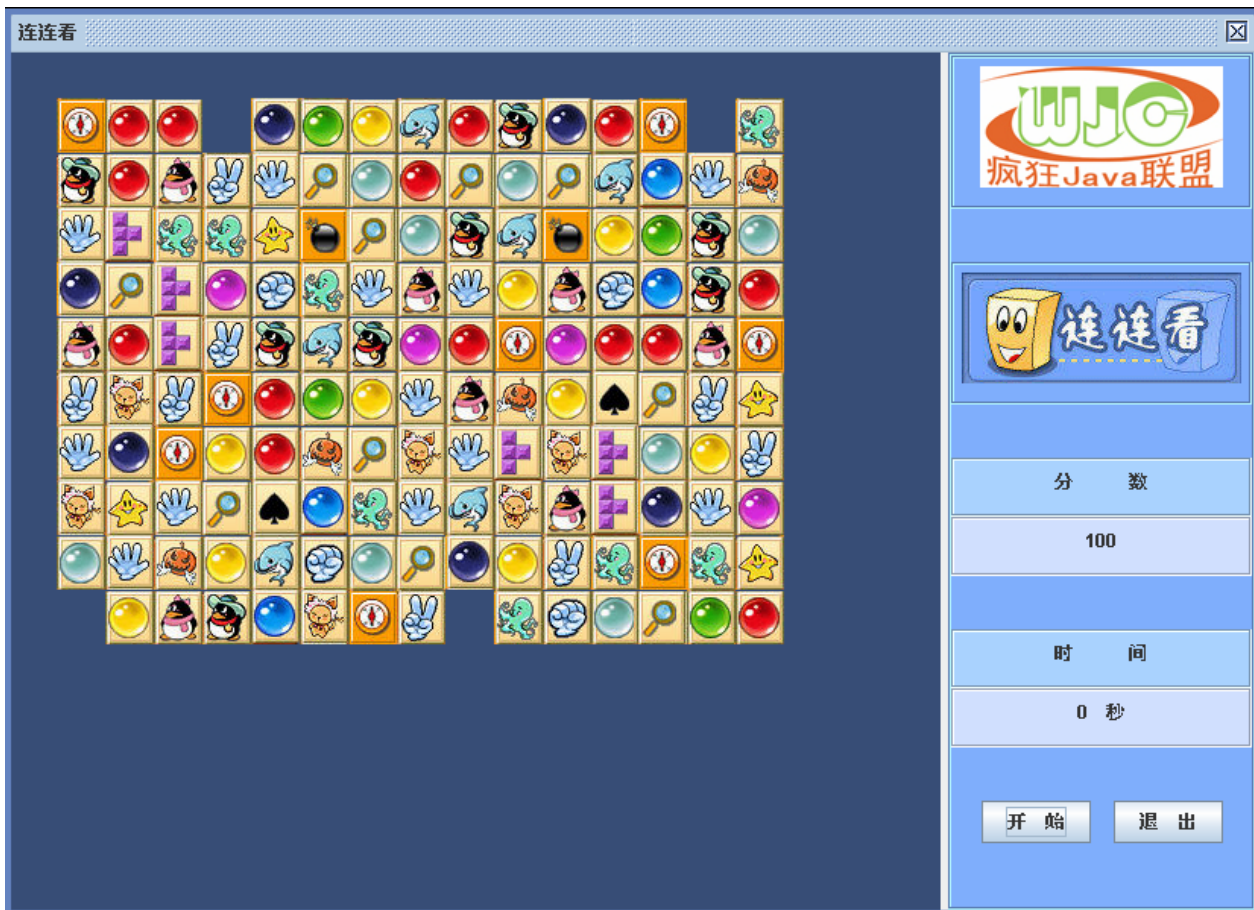


图 7.16 横向两个转折点的相连

下面再去实现第二种情况，纵向的两个转折点的情况，直接在工具方法 `getLinkPoints` 后面加上判断，说明它们两个 `Point` 在纵向有两个转折点即可。这里直接给出代码。

代码清单：code\linkgame\src\org\crazyit\linkgame\service\impl\GameServiceImpl.java

```

if (isInColumn(point1, point2)) { //在同一列
    //向左遍历
    //以 p1 的中心点向左遍历获取点集合
    List<Point> p1LeftChanel = getLeftChanel(point1, 0, pieceWidth);
    //以 p2 的中心点向左遍历获取点集合
    p2LeftChanel = getLeftChanel(point2, 0, pieceWidth);
    Map<Point, Point> leftLinkPoints = getYLinkPoints(p1LeftChanel,
        p2LeftChanel, pieceWidth);
    //向右遍历, 不得超过游戏区域的边框
    //以 p1 的中心点向右遍历获取点集合
    p1RightChanel = getRightChanel(point1, widthMax, pieceWidth);
    //以 p2 的中心点向右遍历获取点集合
    List<Point> p2RightChanel = getRightChanel(point2, widthMax, pieceWidth);
    Map<Point, Point> rightLinkPoints = getYLinkPoints(p1RightChanel,
        p2RightChanel, pieceWidth);
    result.putAll(leftLinkPoints);
    result.putAll(rightLinkPoints);
}

```

注意以上代码中的 2 个 Map，代表着向左与向右两种连接情况。效果如图 7.17 所示。



图 7.17 实现纵向两个转折点连接

纵向两个转折点和横向两个转折点一样，图 7.18 说明为什么需要这样实现。

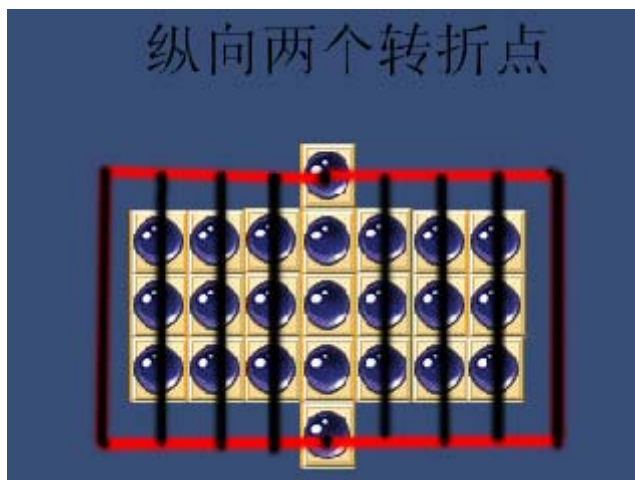


图 7.18 代码说明

图 7.18 中的黑色竖线两端，就是我们结果 Map 中保存的 key 和 value，如果这 key 和 value 这两个点之间没有障碍的话，就加到结果的 Map 中去，如果有障碍，将不会加到结果的 Map 中。下面实现两个转折点，图 7.19 至图 7.24 讲解两个转折点的情况。

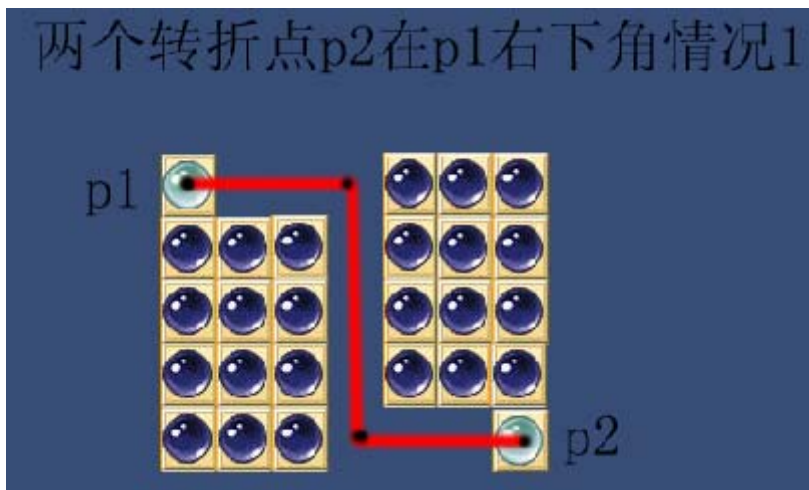


图 7.19 两个转折点 p2 在 p1 右下角的情况 1

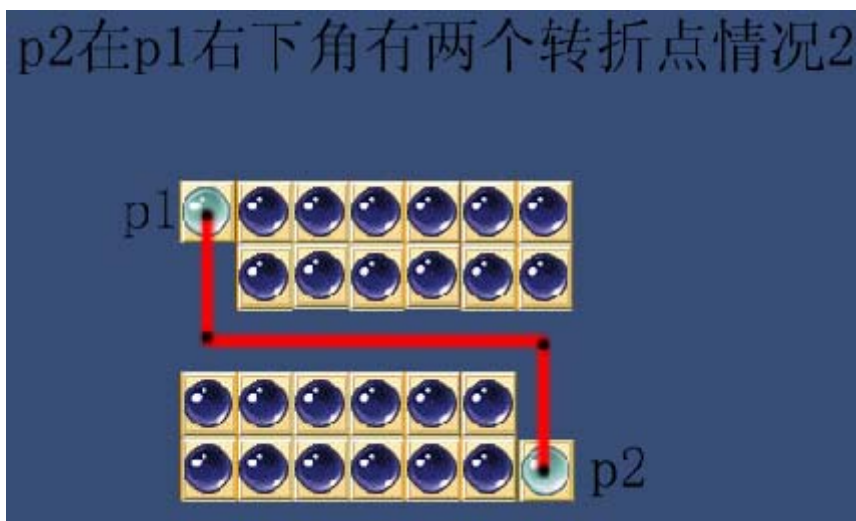




图 7.20 两个转折点 p2 在 p1 右下角的情况 2

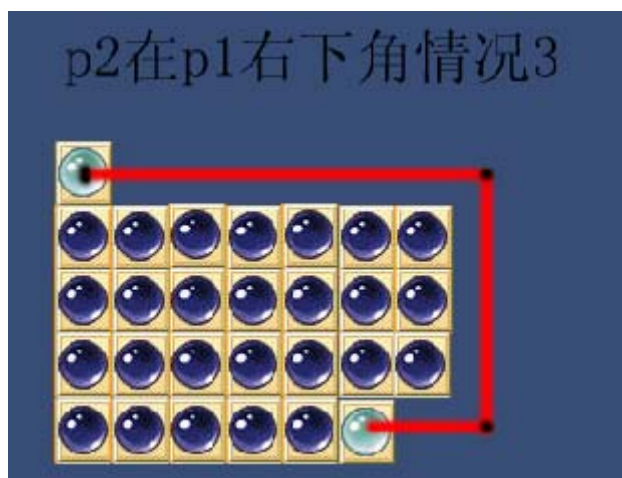


图 7.21 两个转折点 p2 在 p1 右下角的情况 3

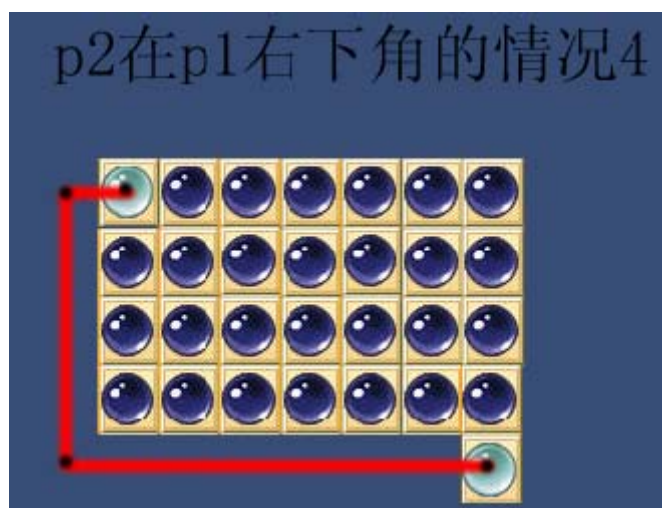


图 7.22 两个转折点 p2 在 p1 右下角的情况 4

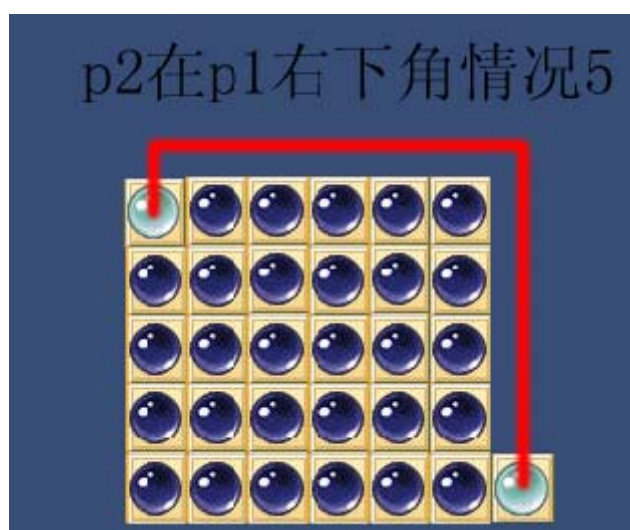


图 7.23 两个转折点 p2 在 p1 右下角的情况 5

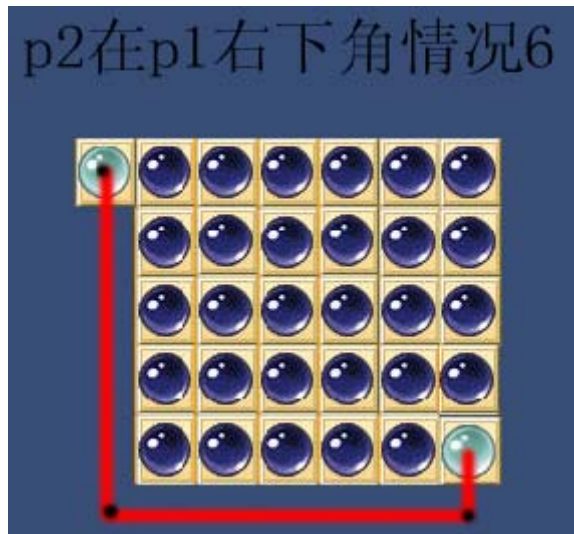


图 7.24 两个转折点 p2 在 p1 右下角的情况 6

实现的代码如下。

代码清单：code\linkgame\src\org\crazyit\linkgame\service\impl\GameServiceImpl.java

```
//获取 point1 向下遍历, point2 向上遍历时横向可连接的点
Map<Point, Point> downUpLinkPoints = getXLinkPoints(p1DownChanel, p2UpChanel, pieceWidth);
//获取 point1 向右遍历, point2 向左遍历时纵向可连接的点
Map<Point, Point> rightLeftLinkPoints = getYLinkPoints(p1RightChanel, p2LeftChanel, pieceHeight);
//获取以 p1 为中心的向上通道
p1UpChanel = getUpChanel(point1, 0, pieceHeight);
//获取以 p2 为中心的向上通道
p2UpChanel = getUpChanel(point2, 0, pieceHeight);
//获取 point1 向上遍历, point2 向上遍历时横向可连接的点
Map<Point, Point> upUpLinkPoints = getXLinkPoints(p1UpChanel, p2UpChanel, pieceWidth);
//获取以 p1 为中心的向下通道
p1DownChanel = getDownChanel(point1, heightMax, pieceHeight);
//获取以 p2 为中心的向下通道
p2DownChanel = getDownChanel(point2, heightMax, pieceHeight);
//获取 point1 向下遍历, point2 向下遍历时横向可连接的点
Map<Point, Point> downDownLinkPoints = getXLinkPoints(p1DownChanel, p2DownChanel, pieceWidth);
//获取以 p1 为中心的向左通道
List<Point> p1LeftChanel = getLeftChanel(point1, 0, pieceWidth);
//获取以 p2 为中心的向左通道
p2LeftChanel = getLeftChanel(point2, 0, pieceWidth);
//获取 point1 向左遍历, point2 向左遍历时纵向可连接的点
Map<Point, Point> leftLeftLinkPoints = getYLinkPoints(p1LeftChanel, p2LeftChanel, pieceHeight);
//获取以 p1 为中心的向右通道
p1RightChanel = getRightChanel(point1, widthMax, pieceWidth);
//获取以 p2 为中心的向右通道
List<Point> p2RightChanel = getRightChanel(point2, widthMax, pieceWidth);
//获取 point1 向右遍历, point2 向右遍历时纵向可以连接的点
Map<Point, Point> rightRightLinkPoints = getYLinkPoints(p1RightChanel, p2RightChanel, pieceHeight);
result.putAll(downUpLinkPoints);
```



```
result.putAll(rightLeftLinkPoints);//将可以连接的所有点都放到结果中
result.putAll(upUpLinkPoints);
result.putAll(downDownLinkPoints);
result.putAll(leftLeftLinkPoints);
result.putAll(rightRightLinkPoints);
```

注：上面黑体的几个 Map，就是代表图 7.19 到图 7.24 的 6 种情况。效果如图 7.25 所示。



图 7.25 实现 p2 在 p1 的右下角两个转折点的连接

下面再实现 p2 在 p1 右上角的情况，这种情况和 p2 在 p1 右下角的情况类似，都是有 6 种情况，实现过程这里不再详细描述。所有的连接方式已经全部实现了，我们可以去看游戏的效果，体验一下初步的游戏成果。

### 7.4.8 找出最短距离

我们在前面实现的时候，只是返回结果 `map` 中的第一个元素，现在，我们需要从这些元素中找出最短的距离返回，原来的代码如下。

代码清单: `code\linkgame\src\org\crazyit\linkgame\service\impl\GameServiceImpl.java`

```
//该 map 的 key 存放第一个转折点, value 存放第二个转折点, map 的 size 说明有多少个可以连的方式
Map<Point, Point> turns = getLinkPoints(p1Point, p2Point, pieceWidth, pieceHeight);
if (turns.size() != 0) {
    for (Object turn : turns.keySet()) {
```

```

        //遍历该 map, 暂时只返回 map 中第一个元素
        //获取 map 中第一个元素的 key 值, 即第一个转折点
        Point point1 = (Point)turn;
        //获取 map 中第一个元素的 value 值, 即第二个转折点
        Point point2 = turns.get(point1);
        return new LinkInfo(p1Point, point1, point2, p2Point);
    }
}

```

现在, 我们对这些代码作些修改, 让它返回四个点 (两个选择的点, 两个转折点) 之间最短的距离。

代码清单: code\linkgame\src\org\crazyit\linkgame\service\impl\GameServiceImpl.java

```

//获取 p1 和 p2 之间最短的连接信息
private LinkInfo getShortcut(Point p1, Point p2, Map<Point, Point> turns,
    int shortDistance) {
    List<LinkInfo> infos = new ArrayList<LinkInfo>();
    //遍历结果 map, 将转折点与选择的点封装成 LinkInfo 对象, 放到集合中
    for (Object info : turns.keySet()) {
        Point point1 = (Point)info;
        Point point2 = turns.get(point1);
        infos.add(new LinkInfo(p1, point1, point2, p2));
    }
    return getShortcut(infos, shortDistance);
}
//在 infos 中获取其四个点最短的那个 LinkInfo 对象
private LinkInfo getShortcut(List<LinkInfo> infos, int shortDistance) {
    int temp1 = 0;
    LinkInfo result = null;
    for (int i = 0; i < infos.size(); i++) {
        LinkInfo info = infos.get(i);
        //计算出几个点的总距离
        int distance = countAll(info.getLinkPoints());
        //将循环第一个的差距用 temp1 保存
        if (i == 0) {
            temp1 = distance - shortDistance;
            result = info;
        }
        //如果下一次循环的值比 temp1 的还小, 则用当前的值作为 temp1
        if (distance - shortDistance < temp1) {
            temp1 = distance - shortDistance;
            result = info;
        }
    }
    return result;
}
//计算 points 中所有点的距离总和
private int countAll(List<Point> points) {
    int result = 0;
    for (int i = 0; i < points.size(); i++) {
        if (i == points.size() - 1) { //循环到最后一个
            break;
        }
    }
}

```

```

        Point point1 = points.get(i);
        Point point2 = points.get(i + 1);
        result += getDistance(point1, point2);
    }
    return result;
}
//获取两个 LinkPoint 之间的最短距离
private int getDistance(Point p1, Point p2) {
    int xDistance = Math.abs(p1.getX() - p2.getX());
    int yDistance = Math.abs(p1.getY() - p2.getY());
    return xDistance + yDistance;
}

```

实现原理：遍历转折点 **map** 中的所有转折点，与原来选择的两个点构成一个 **LinkInfo**，再加入一个集合中，再去遍历这个集合，选取最接近最短距离的那一个 **LinkInfo** 返回即可。

### 7.4.9 画上连接线

现在我们得到了 **LinkInfo** 对象，之前并没有对 **LinkInfo** 对象进行处理，现在可以在 **GamePanel** 中为 **LinkInfo** 中的 **Point** 对象画上相应的连接线，为 **LinkInfo** 画上连接线。

代码清单：code\linkgame\src\org\crazyit\linkgame\view\GamePanel.java

```

private void drawLine(LinkInfo linkInfo, Graphics g) {
    List<Point> points = linkInfo.getLinkPoints();
    for (int i = 0; i < points.size() - 1; i++) {
        Point currentPoint = points.get(i);
        Point nextPoint = points.get(i + 1);
        g.drawLine(currentPoint.getX(), currentPoint.getY(),
            nextPoint.getX(), nextPoint.getY());
    }
}

```

在 **paint** 方法中调用上面的工具方法：

```

//如果当前对象中有 linkInfo 对象，即连接信息
if (this.linkInfo != null) {
    drawLine(this.linkInfo, g);
    //处理完后清空 linkInfo 对象
    this.linkInfo = null;
}

```

## 7.5 加入计分与计时功能

### 7.5.1 加入计分功能

玩家点击了游戏区域时，程序中需要调用 **GameService** 的 **start** 方法，我们需要为开始按钮加入鼠标监听器，我们可以将这个监听器单独作为一个类，新建 **BeginListener**，该类继承于 **MouseListenerAdapter**，只要重写父类的 **mousePressed** 方法即可。

代码清单：code\linkgame\src\org\crazyit\linkgame\listener\BeginListener.java

```

public void mousePressed(MouseEvent e) {
    if (this.timer != null) {

```

```
        this.timer.cancel();
    }
    this.timer = new Timer();
    gamePanel.setSelectPiece(null);
    gamePanel.setOverImage(null);
    // 将分数清 0
    pointLabel.setText("0");
    // 将时间变成原来设置的时间(在 GameConfiguration 中设置)
    timeLabel.setText(String.valueOf(config.getGameTime()));
    // 调用 gameService 的 start 方法
    gameService.start();
    // 开始进行任务
    task = new TimerTask(this.gamePanel, this.config.getGameTime(),
        this.timeLabel);
    timer.schedule(task, 0, 1000);
    // 对 gamePanel 进行重新绘制
    gamePanel.repaint();
}
```

**BeginListener** 中的鼠标点击方法较为简单, 开始时只要设置分数、时间, 并调用 **GameService** 的 **start** 方法即可。

实现普通的计分功能比较简单, 可以在 **GameListener** (**GamePanel** 的鼠标监听器) 中判断, 当成功连接了之后, 就加上一定的分数, 并设置到 **GameService** 中, 最后把结果显示到 **GamePanel** 的 **JLabel** 中。**GameService** 中添加一个接口方法, 让 **GameServiceImpl** 去实现该方法。

代码清单: code\linkgame\src\org\crazyit\linkgame\service\impl\GameServiceImpl.java

```
private long grade = 0; //加入分数属性,初始值为 0
public long countGrade() {
    this.grade += this.config.getPerGrade();
    return this.grade;
}
```

效果图 7.26 所示。



图 7.26 加入计分功能

### 7.5.2 加入计时功能与游戏的胜利、失败

现在实现连连看的计时功能，本例的计时功能采用倒数模式，当时间到了 0 后，在游戏区内如果还有图片方块的话，游戏就失败。

先为我们的游戏加入胜利和失败的判断，胜利和失败的提示，是在 `GamePanel` 中提示的，因此，在 `GamePanel` 对象中添加一个 `overImage` 的属性，用于保存游戏胜利或者失败时图片，并以此作为游戏胜利和失败的标准，并在 `GamePanel` 中加入画胜利和失败图片。

代码清单：code\linkgame\src\org\crazyit\linkgame\view\GamePanel.java

```
//如果 overImage 不为空，则表示游戏已经胜利或者失败  
if (this.overImage != null) g.drawImage(this.overImage, 0, 0, null);
```

这时候，我们需要知道哪里需要时间计算的，首先肯定是游戏开始时时间开始计算，当游戏区中已经没有方块时，即最后一点成功的连接后，时间计算停止，因此 `7.5.2BeginListener` 类与游戏区的监听类 `GameListener` 都必须可以控制时间。

先编写一个 `TimerTask` 类，用于定时执行任务。

代码清单：code\linkgame\src\org\crazyit\linkgame\timer\TimerTask.java

```
public class TimerTask extends java.util.TimerTask {  
    private long time; //当前用掉的时候  
    private GamePanel gamePanel;  
    private long gameTime;  
    private JLabel timeLabel;
```

```

public void run() {
    //游戏时间已到
    if (this.gameTime - this.time <= 0) {
        //设置游戏图片为失败
        this.gamePanel.setOverImage(ImageUtil.getImage("images/lose.gif"));
        //取消这个任务
        this.cancel();
        this.gamePanel.repaint();
    }
    //如果游戏仍然继续, 设置时间
    this.timeLabel.setText(String.valueOf(this.gameTime - this.time));
    this.timeLabel.repaint();
    //使用的时间+1
    this.time += 1;
}
}

```

该类继承 `java.util.TimerTask`, 实现 `run` 方法, 表示需要执行的动作。在 `TimerTask` 的构造器中, 需要将 `gamePanle`, `gameTime`, `timeLabel` 作为参数传入, 首先 `timeLabel` 是时间显示的一个 `JLabel`, 因此必须加入, `gameTime` 我们保存在 `GameConfiguration` 对象中, `gamePanel` 中保存了游戏胜利与失败的标准。另外再为 `TimerTask` 类添加一个用掉的时间属性, 用于记录游戏已经用掉的时间, 并在 `run` 方法中加一, 表示 `run` 每执行一次, `time` 的值加一, 再用参数的 `gameTime` 减去 `time` 属性的值, 就是游戏所剩的时间, 当游戏所剩的时间为 0 时, 游戏失败, 计时器停止。

在 `GameListener` 中如果成功连接了两个 `Piece` 对象后, 可以判断游戏区是否还有方块的存在, 即判断该数组中象是的每一个 `Piece` 是否为空, 如果都为空, 则游戏胜利, 在 `GameServiceImpl` 类中加入如下 `hasPieces` 方法, 用于判断界面中是否还存在图片。

代码清单: `code\linkgame\src\org\crazyit\linkgame\service\impl\GameServiceImpl.java`

```

//实现接口的 hasPieces 方法
public boolean hasPieces(Piece[][] pieces) {
    for (int i = 0; i < pieces.length; i++) {
        for (int j = 0; j < pieces[i].length; j++) {
            if (pieces[i][j] != null) return true;
        }
    }
    return false;
}

```

这样, 当成功点击连接最后两个方块后, 就停止计时器, 并设置游戏为胜利状态, 并在 `GameListener` 的 `mousePressed` 方法最前面加入, 当游戏胜利或者, 去点击游戏区, 就马上返回, 不再作任何判断:

```

if (gamePanel.getOverImage() != null) return;

```

好了, 计时功能已经实现了, 现在可以去看下游戏效果如图 7.27 所示。



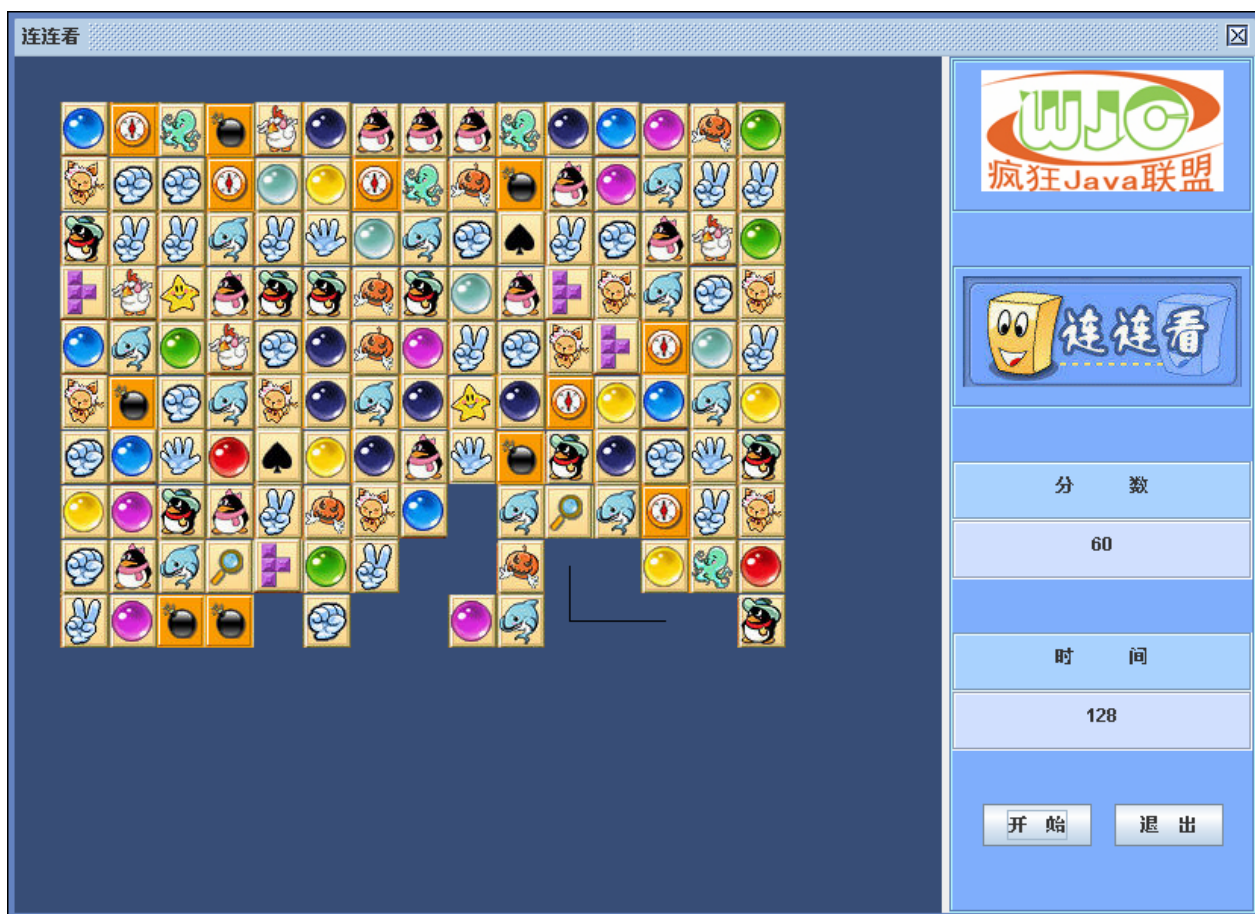


图 7.27 最终的游戏效果

## 7.6 本章小结

本章主要讲述如何实现一个简单的单机版连连看，详细介绍了如何建立界面，如何将游戏的区域抽象成一个二维数组，并对该二维数组进行操作，实现了随机游戏数组，实现游戏的相连，游戏的计分功能与计时等功能。重点介绍了如何在重构代码的过程中优化代码并慢慢的改善既有设计，将游戏的部件或者各部件的职能抽象成对象。在本章中代码还可以进行重构优化，例如将每个图形界面组件做成单独的类，再通过工厂模式或者 Java 的反射去获取这些类，用于去创建游戏的界面。

## 第 8 章 简单Java IDE工具

### 8.1 IDE工具简介

IDE 是 Integrated Development Environment 的缩写，即集成开发环境，就是集成了代码编写功能、分析功能、编译功能、debug 功能等一体化的开发套件。例如 Java 程序员经常使用的像 Eclipse、NetBeans、JBuilder 等这些功能强大的 IDE 工具。

### 8.2 Java IDE的主要功能

本章我们主要编写一个简单的 Java IDE 工具，功能并不多，实现文件操作，文本操作，项目管理，Java 文件的编译和运行等基本功能即可。一个真正的 Java IDE 还包括个性化设置，代码的生成，打点提示，debug，文件搜索等一系列附加的功能，这些功能都大大的方便了程序员的开发工作。在本章中我们并不需要实现像 Eclipse 这样的优秀工具，只需要将去实现自己的一个拥有简单功能的 IDE，让我们了解如何去实现这些十分常见的功能。

### 8.3 建立界面

在开发前，我们需要知道一个 IDE 有些什么界面，需要显示一些什么，注意我们在编写程序的过程中需要小步快跑，如果一开始定的目标越大，那么往往做到一定程度时会有些不知所措。我们需要建立的界面确定为图 8.1 的布局。建立好这个布局，我们还需要创建菜单，建立工作空间选择界面，项目、文件、目录的新建界面。

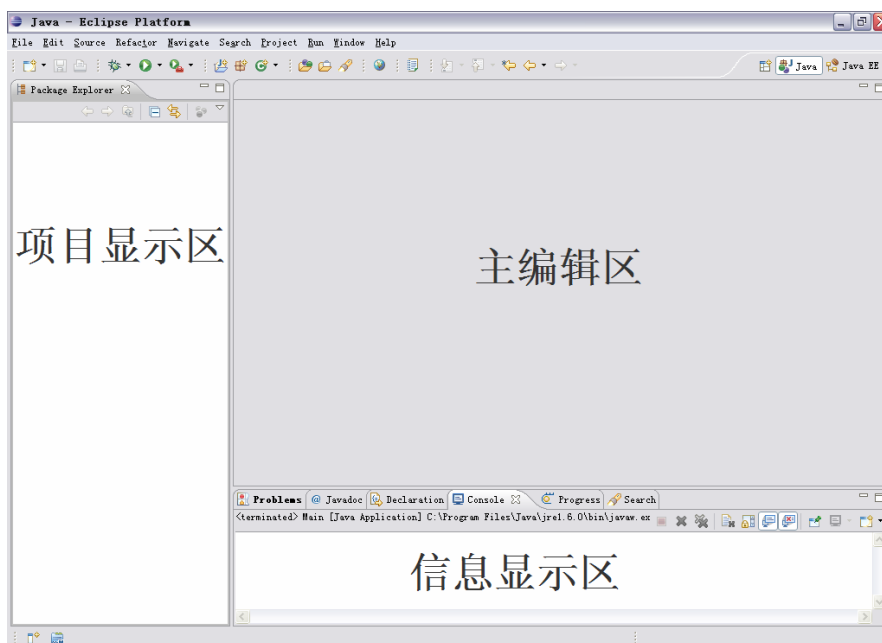


图 8.1 确定界面布局

图 8.1 借用了 Eclipse 的界面，Eclipse 中的每个区域都可以自由显示与关闭，我们在本例中并不需要做到那么复杂，只需要做到以上的布局，并可以自由的拖动每个区的大小即可，这是我们的第一个目标。

### 8.3.1 建立主编辑区和信息显示区

按照图 8.1，我们需要先建立主界面的 **JFrame**，并往里面添加我们所需要的组件。在本例中，我们需要做到多文件编辑，因此需要 **JDesktopPane**，因此，我们的主编辑中主要存放一个 **Box** 对象，**Box** 对象中放一个 **JTabbedPane** 作为 **tab** 页签，再存放一个 **JDesktopPane** 对象。信息显示区主要是一个不可编辑的 **JTextArea** 对象。新建程序入口 **Main** 类，作为这个 IDE 工具的入口。

代码清单：code\editor\src\org\crazyit\editor\Main.java

```
public class Main {
    public static void main(String[] args) {
        EditorFrame editorFrame = new EditorFrame("ide");
        editorFrame.setVisible(true);
    }
}
```

再新建一个界面的 **JFrame** 类。代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```
public class EditorFrame extends JFrame {
    public EditorFrame(String title) {
        super(title); //设置标题
        pack(); //使 JFrame 调整最佳大小
    }
}
```

建立完 **EditorFrame** 后，我们需要往这个 **JFrame** 里面添加组件，让其符合我们的要求。添加组件我们可以新建一个 **initFrame** 的方法，该方法主要用来创建这个 **JFrame** 里面的各个组件。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```
private JTabbedPane tabPane; //多文件的 tab 标题
private Box box; //存放 tabPane 与 desk
private JDesktopPane desk; //创建一个多文档的桌面容器
private JSplitPane editorSplitPane; //用于分隔主编辑区和信息显示区的容器
private JScrollPane infoPane; //可以滚动的 JScrollPane 对象，用于放 infoArea
private JTextArea infoArea; //用于显示信息的文本域
public void initFrame() {
    //设置窗口关闭，退出程序
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    //创建主编辑区的 tabPane
    tabPane = new JTabbedPane(JTabbedPane.TOP, JTabbedPane.SCROLL_TAB_LAYOUT);
    desk = new JDesktopPane(); //创建 JDesktopPane 对象
    desk.setBackground(Color.GRAY); //设置 desk 的背景颜色为灰色
    box = new Box(BoxLayout.Y_AXIS); //设置 box 的布局
    box.add(tabPane);
    box.add(desk);
    infoArea = new JTextArea("", 5, 50); //创建信息显示区的文本域
    infoPane = new JScrollPane(infoArea); //将 infoArea 文本域作为组件放到 infoPane 中
    infoArea.setEditable(false); //设置信息区不可编辑
    //创建这个分隔组件的容器，并将 box 对象和 infoPane 放置其中
```

```
editorSplitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT, box, infoPane);
editorSplitPane.setDividerSize(3);
editorSplitPane.setDividerLocation(500);
add(editorSplitPane);
pack();//使 JFrame 调整最佳大小
}
```

以上代码创建了主编辑区和信息显示区，主编辑区主要是一个 **Box** 对象，该对象里面有 **JTabbedPane** 和 **JDesktopPane**，运行 **Main** 可看到效果，对其进行拖动，可以看到的效果如图 8.2 所示。



图 8.2 创建主编辑区和信息显示区

信息显示区的作用主要是用来显示一些操作的信息，例如编译 **Java** 源文件出错，运行类打印的一些信息等。在编辑区主的 **Box** 对象中，还有一个 **tab** 标签页对象，现在新建了界面，在本章会实现该 **JTabbedPane** 对象的创建（打开），切换等功能。

### 8.3.2 建立项目显示区

项目显示区，可以理解成是一个文件系统树，用于存放一棵文件树，树的每个节点都代表一份文件（目录）。在以下代码中创建项目显示区，暂时创建一棵样例树代替我们的文件树。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```
private JScrollPane treePane; //存放树的可滚动容器
private JSplitPane mainSplitPane; //整个界面的分隔组件的容器
private JTree tree; //项目树对象
public void initFrame() {
    ...
    tree = new JTree();//创建树对象
    treePane = new JScrollPane(tree); //创建可滚动的容器对象
    //创建主界面的 JSplitPane，横向，左边为 treePane，右边为 editorSplitPane
    mainSplitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, treePane,
        editorSplitPane);
    mainSplitPane.setDividerLocation(200); //设置分隔条的位置
    mainSplitPane.setDividerSize(3); //设置分隔条的粗细
}
```

以上代码新建了一个 **JScrollPane** 用于存放我们的项目显示区中的文件树，我们暂时使用 **JTree** 的

示例树代替文件树。运行可看到的效果如图 8.3。

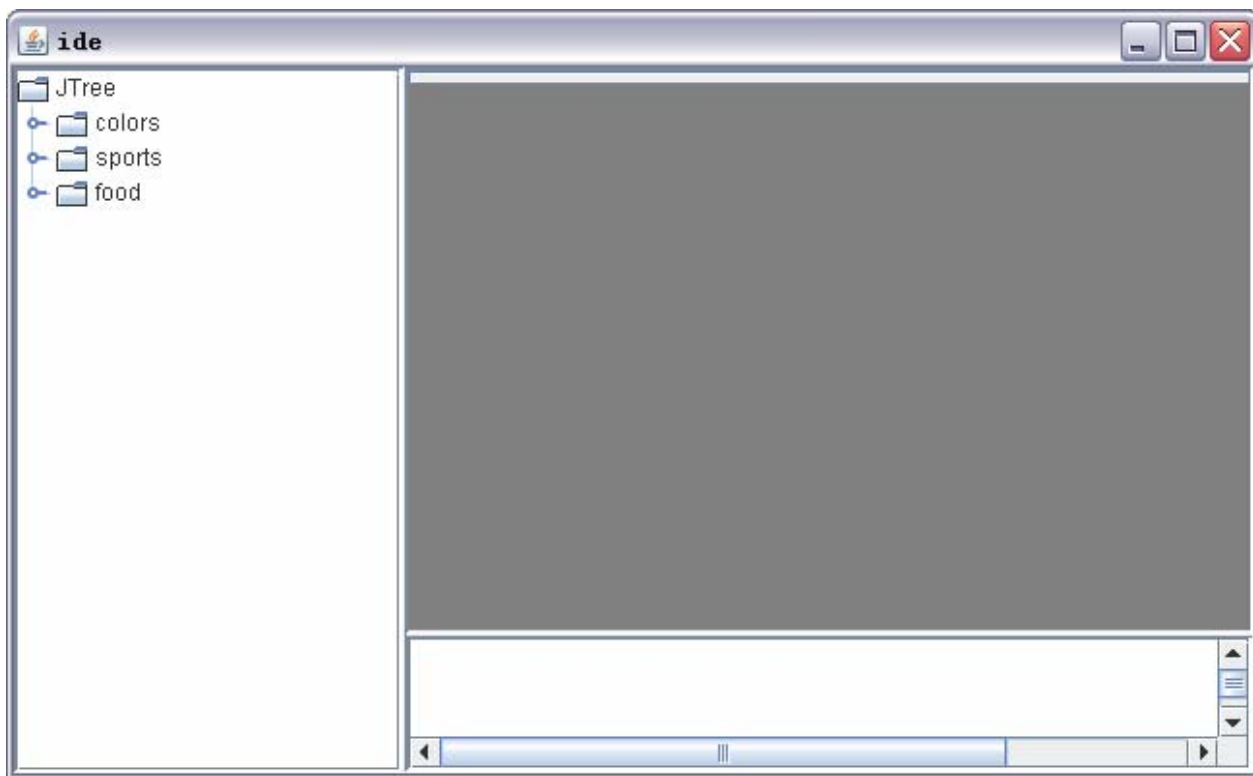


图 8.3 添加了项目显示区后的界面

### 8.3.3 添加菜单和工具栏

在 8.3.2 中建立了主界面的布局后，接下来为这个 IDE 工具添加菜单和工具栏，并为它们添加监听器。主要的文件操作菜单有新建（文件、目录和项目）、打开、保存、刷新、运行和退出。文本操作菜单有复制、粘贴和剪切。工具栏的功能主要有新建文件、打开、保存、刷新、运行、复制、粘贴和剪切。

下面先创建文件的新建、保存、运行、打开，文本的复制、剪切和粘贴，并添加监听器。在 `EditorFrame` 类中新建一个 `addListeners` 方法，用于添加 `EditorFrame` 中各个组件的监听器。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```
private JMenuBar menuBar; //菜单栏对象
private JMenu editMenu; //编辑菜单对象
private JMenu fileMenu; //文件菜单
//新建文件的 Action 对象
private Action fileNew = new AbstractAction("新建文件", new ImageIcon("images/newFile.gif")) {
    public void actionPerformed(ActionEvent e) {
        //暂时提供空实现
    }
};
//省略其他 Action 的创建
//为 EditorFrame 中的组件添加监听器
public void addListeners() {
    //新建文件的监听器
    fileMenu.add(fileNew).setAccelerator(KeyStroke.getKeyStroke('N', InputEvent.CTRL_MASK));
    //省略其他创建监听器的代码
}
```

```

fileMenu.add(exit);
//添加复制监听器
editMenu.add(copy).setAccelerator(KeyStroke.getKeyStroke('C', InputEvent.CTRL_MASK));
//省略添加其他监听器的代码
}
public void initFrame() {
    ...
    menuBar = new JMenuBar();//创建菜单栏对象
    editMenu = new JMenu("编辑");//创建编辑菜单对象
    fileMenu = new JMenu("文件");//创建文件菜单
    menuBar.add(fileMenu); //将文件菜单添加到菜单栏中
    menuBar.add(editMenu); //将编辑菜单添加到菜单栏中
    setJMenuBar(menuBar); //设置 JFrame 的菜单栏
    addListeners();
}

```

注：以上代码中只有新建文件的 Action 对象，其他的 Action 对象的创建方法与其一致，除了退出菜单需要实现退出外，其他 Action 对象的 actionPerformed 的方法体都提供空实现。

在 addListeners 方法，为 fileMenu 添加各个 Action 对象，并有设置了对应的快捷键。运行的效果如图 8.4 所示。



图 8.4 添加文件菜单与编辑菜单

接下来，为 IDE 工具添加工具栏，并为工具栏中的各个功能设置对应的 Action 对象。使用 swing 中的 JToolBar 类可实现工具栏，并为 JToolBar 设置对应的 Action 即可。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```
//工具条
```



```
private JToolBar toolBar;  
public void initFrame() {  
    ...  
    toolBar = new JToolBar();  
    toolBar.setFloatable(false); // 设置工具栏不可移动  
    toolBar.setMargin(new Insets(0, 10, 5, 5)); // 设置工具栏的边距  
    add(toolBar, BorderLayout.NORTH); // 将工具栏添加到 EditorFrame 中  
    ...  
}  
  
// 为 EditorFrame 中的组件添加监听器  
public void addListeners() {  
    ...  
    // 为工具条添加各个操作  
    toolBar.add(fileNew).setToolTipText("新建文件");  
    // 按照该方法依次添加打开、保存、刷新、运行、复制、剪切和粘贴  
    ...  
}
```

重新运行 Main 类，可看到效果如图 8.5 所示。

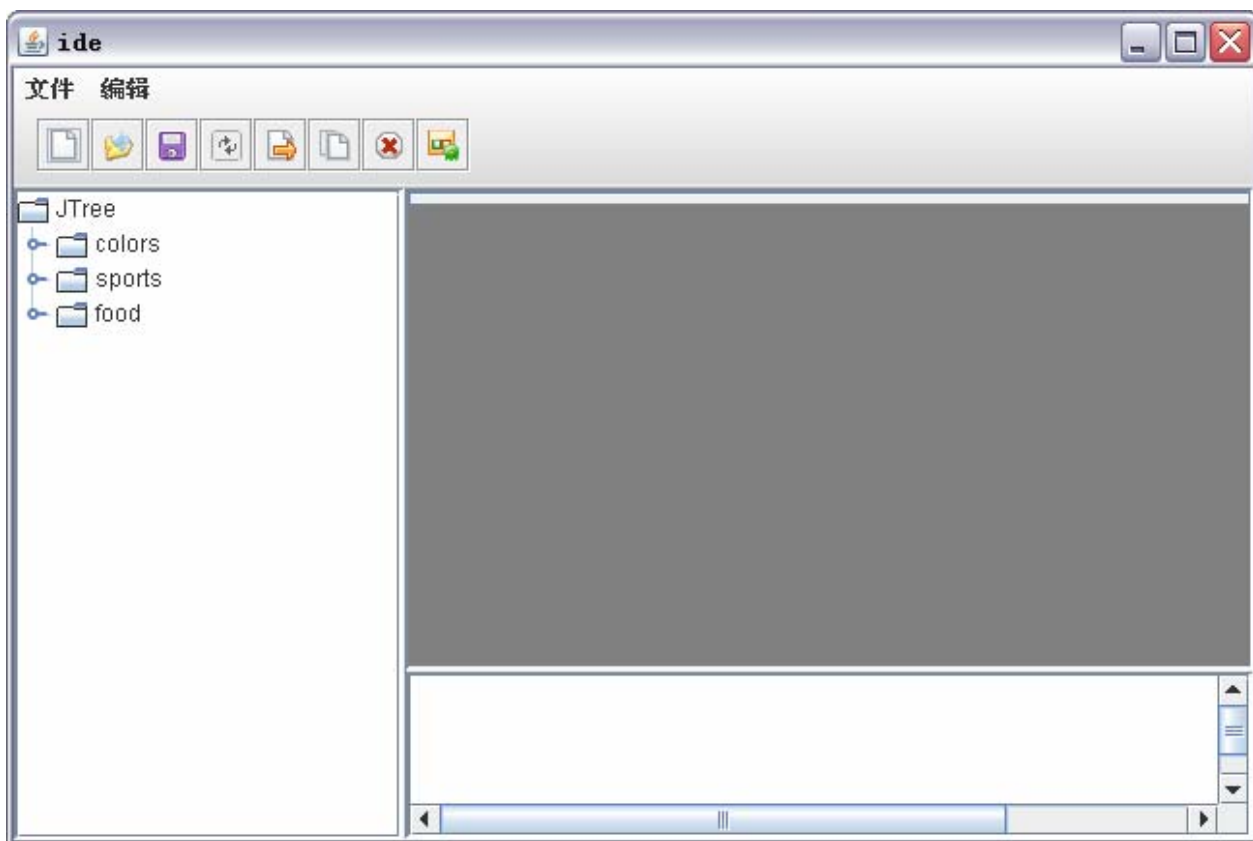


图 8.5 添加工具栏

### 8.3.4 建立工作空间选择界面

到现在，主编辑区的布局与菜单都建好了，之前所定下的两个小目标都已经实现，在进入这个界面

之前，我们需要有一个界面让用户去设置项目存放的目录，即工作空间。现在我们先定下一个小目标，建立一个选择工作空间的界面，让用户在使用这个 IDE 时选择工作空间，做到的效果大概如图 8.6 所示。

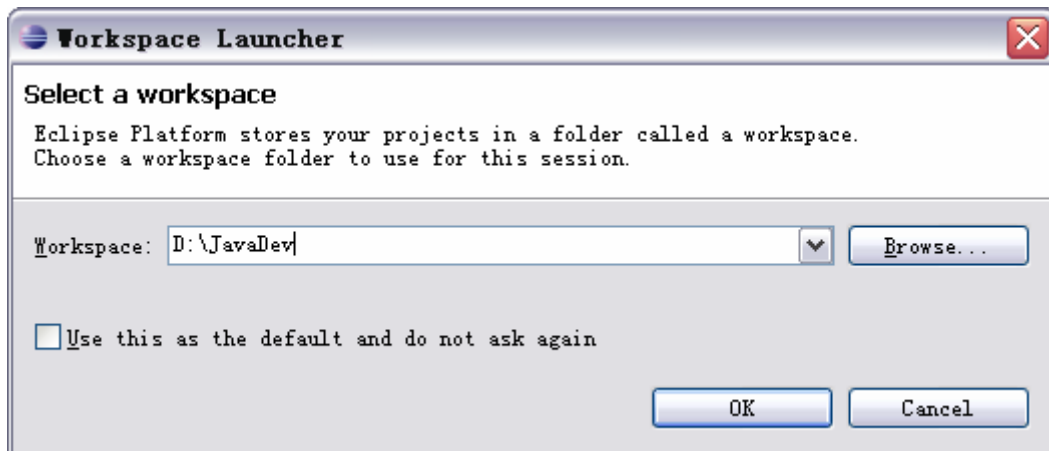


图 8.6 Eclipse 中选择工作空间的界面

图 8.6 中借用了 Eclipse 中选择工作空间的界面，我们所定的界面只需要有一个文件选择器，让用户去选择工作空间的目录，并有一个文本框显示该目录的全路径，再有确定和取消的按钮。我们创建一个 `SpaceFrame` 的类用于建立工作空间选择界面。

注：我们不必做到记录历史工作空间，选定默认的工作空间等功能。

通过新建一个 `JFrame` 来创建该界面即可。图 8.7 创建了一个工作空间选择界面。



图 8.7 创建工作空间选择界面（`SpaceFrame` 类）

注意，图 8.7 中的确定按钮为不可用，由于没有选择工作空间，那么该按钮就不可以点击，使用 `button.setEnabled(false)` 即可达到效果。我们本小节只是创建界面，因为文件选择按钮、确定和取消按钮的事件暂时不用实现。另外，工作空间显示的文本框设置为不可用，不允许用户直接输入字符，只能通过选择按钮选择工作空间。现在，工作空间创建的界面已经实现了。

### 8.3.5 建立文件、目录、项目添加界面

我们在前面已经建立了编辑主界面、选择工作空间界面，现在，还需要一个添加的界面，用于添加文件、目录和项目的界面。该界面只需要一个文本框，用于输入文件名称、目录名称和项目名称，并有确定和取消的按钮，暂时不需要加上事件，所以实现比较简单。实现的效果如图 8.8 所示。



图 8.8 建立添加界面

注意，在这里的这个 `JFrame`，如果关闭的时候，只是普通的设置为不显示，而不是退出程序。

代码清单：code\editor\src\org\crazyit\editor\AddFrame.java

```
private JPanel mainPanel; //该 Frame 的 JPanel
private JPanel namePanel; //项目名称
private JLabel nameLabel; //显示文件的 JLabel
private JTextField nameText; //输入名称的 JTextField
private JPanel buttonPanel; //放按钮的 Panel
private JButton confirmButton; //确定按钮
private JButton cancelButton; //取消按钮
public AddFrame() {
    //创建一系列界面组件的代码
    mainPanel = new JPanel();
    namePanel = new JPanel();
    nameLabel = new JLabel("文件名称: ");
    nameText = new JTextField("", 20);
    buttonPanel = new JPanel();
    confirmButton = new JButton("确定");
    cancelButton = new JButton("取消");
    //设置界面布局的代码
    mainPanel.setLayout(new BoxLayout(mainPanel, BoxLayout.Y_AXIS));
    setLocation(200, 200);
    setResizable(false);
    //设置存放文本框的 JPanel 的布局
    namePanel.setLayout(new BoxLayout(namePanel, BoxLayout.X_AXIS));
    namePanel.add(nameLabel);
    namePanel.add(nameText);
    //确定和取消的按钮
    buttonPanel.setLayout(new BoxLayout(buttonPanel, BoxLayout.X_AXIS));
    confirmButton.setEnabled(false);
    buttonPanel.add(confirmButton);
    buttonPanel.add(new JLabel("  "));
    buttonPanel.add(cancelButton);
    //将文本框的 JPanel 和按钮的 JPanel 添加到 mainPanel 中
    mainPanel.add(namePanel);
    mainPanel.add(buttonPanel);
    //将 mainPanel 添加到 JFrame 中
    add(mainPanel);
    pack();
}
```

现在创建了一个添加的界面，可能会觉得奇怪，需要添加文件、目录、项目，需要三个界面，而这里只建立了一个界面。实际上，添加文件、目录、项目可以共用这个界面，这个界面的作用只是获取用户输入的名称，再根据这个名称作处理，在下面章节，将会用这个界面做不同的事。

## 8.4 实现工作空间选择功能

在做主界面前，我们需要用户选择一个目录作为工作空间，就好像 `Eclipse` 那样，进入时选择工作空间。在前 8.3.4 小节，我们已经创建了选择工作空间的界面，如图 8.7 所示。那么现在，我们就围绕这个界面做工作空间选择功能。

首先，修改程序入口类，程序启动，显示选择工作空间的界面，并将之前创建的 `EditorFrame` 对象变为不可见，并将该对象当作参数传给工作空间的 `JFrame` 类（`SpaceFrame` 类）

代码清单：code\editor\src\org\crazyit\editor\Main.java

```
public static void main(String[] args) {
    EditorFrame editorFrame = new EditorFrame("ide");//创建 EditorFrame，暂时不用设置可见
    //将 editorFrame 对象作为 SpaceFrame 的构造参数
    SpaceFrame spaceFrame = new SpaceFrame(editorFrame);
    spaceFrame.setVisible(true); //让 SpaceFrame 可见
}
```

将 `EditorFrame` 传给 `SpaceFrame`，因为在 `SpaceFrame` 中选择了工作空间（目录）后，我们需要设置 `EditorFrame` 为可见，并告诉 `EditorFrame` 选择了哪个工作空间。运行 `Main` 类，就可以看到效果，只看到工作选择空间的界面。

### 8.4.1 实现目录选择功能

在选择工作空间时，我们需要一个文件选择器用于选择文件，并且只能选择目录，如图 8.8 中的文本框右边的“选择”按钮，当点击这个按钮时，就弹出文件选择器，用于选择某个我们需要作为工作空间的目录。

现在写一个选择按钮的监听器类 `ChoseButtonListener`，可以将该类放到 `SpaceFrame.java` 中，具体代码如下。代码清单：code\editor\src\org\crazyit\editor\SpaceFrame.java

```
class ChoseButtonListener implements ActionListener {
    private JFileChooser chooser; //需要打开的文件选择器对象
    //创建 ChoseButtonListener 需要文件选择器作构造参数
    public ChoseButtonListener(JFileChooser chooser) {
        this.chooser = chooser;
    }
    public void actionPerformed(ActionEvent arg0) {
        //设置该文件选择器只能选择文件目录
        chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
        chooser.showOpenDialog(null); //显示这个文件选择器
    }
}
```

由于 `ChoseButtonListener` 的构造参数中需要一个文件选择器对象，因此我们需要创建一个文件选择器的对象，建立一个 `SpaceChooser` 类，同样可以将该类放到 `SpaceFrame.java` 中，具体代码如下。

代码清单：code\editor\src\org\crazyit\editor\SpaceFrame.java

```
class SpaceChooser extends JFileChooser {
    private SpaceFrame spaceFrame;
    //需要将 SpaceFrame 作为构造参数
    public SpaceChooser(SpaceFrame spaceFrame) {
        super("/");//设置选择器打开时的目录
        this.spaceFrame = spaceFrame;
    }
    //重写父类的选择文件方法
    public void approveSelection() {
        File folder = getSelectedFile();//获取用户选择的文件
        spaceFrame.setFolder(folder); //设置 SpaceFrame 的属性 folder 的值
        spaceFrame.getPathText().setText(folder.getAbsolutePath()); //设置 SpaceFrame 文本框
        spaceFrame.getConfirmButton().setEnabled(true); //设置确定按钮可用
        super.approveSelection(); //调用父类的选择文件方法
    }
}
```

```

    }
}

```

注意以上代码的黑体部分，在 `SpaceFrame` 中，需要添加一个 `File` 的属性，用于存放用户所选择的文件，当用户利用文件选择器选择了文件目录时，需要将 `SpaceFrame` 中的 `JTextFiedl pathText` 的值设置为用户选择的文件的绝对路径，将 `SpaceFrame` 中的确定按钮设置为可用。`SpaceFrame` 中的 `folder` 属性，`pathText` 属性与 `confirmButton` 属性的代码如下。

代码清单：code\editor\src\org\crazyit\editor\SpaceFrame.java

```

//工作空间中显示用户选择文件目录的 JTextField
private JTextField pathText;
//工作空间中的确定按钮
private JButton confirmButton;
//用户选择的文件目录对象
private File folder;

```

这三个属性中，`folder` 属性需要提供 `setter` 和 `getter` 属性，`pathText`（路径显示文本框）和 `confirmButton`（确定按钮）需要提供 `getter` 方法。

创建完文件选择器后，可以 `SpaceFrame` 中的选择按钮添加监听器，该监听器就是上面的 `ChoseButtonListener` 对象，创建的代码如下。

代码清单：code\editor\src\org\crazyit\editor\SpaceFrame.java

```

private JButton choseButton;
private SpaceChooser chooser;
public SpaceFrame(EditorFrame editorFrame) {
    //省略创建界面的各个组件和布局的代码
    ...
    //为 choseButton 添加监听器，并将 SpaceChooser 对象作为构造参数
    choseButton.addActionListener(new ChoseButtonListener(chooser));
    ...
}

```

现在，运行 `Main` 类，点击选择按钮，可看到效果，如图 8.9 所示：

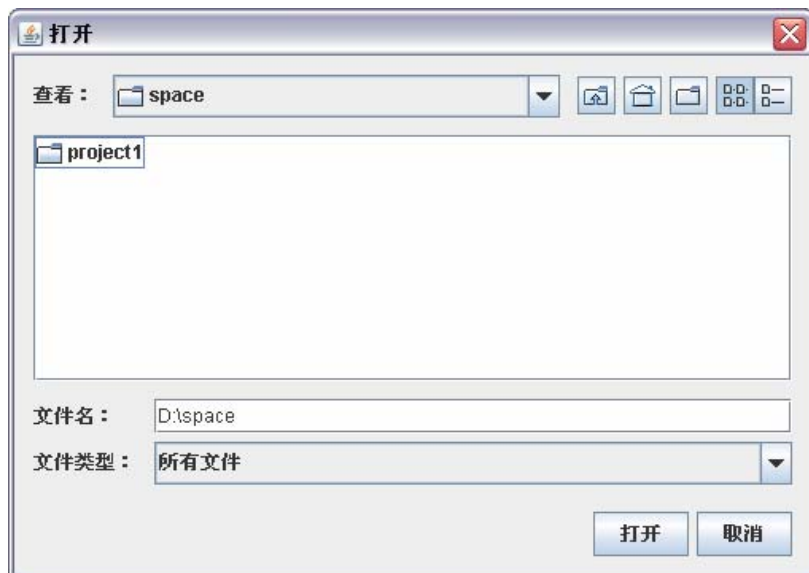


图 8.9 选择工作空间

选择了某个目录进行打开操作，可以看到选择工作目录的界面如图 8.10 所示。



图 8.10 显示选择的工作空间

选择完工作空间后，显示的文本框就出现了所选的目录的绝对路径，确定按钮也变得可用了，这也符合我们的预期效果。

实现了这一步之后，我们还需要实现确定按钮的动作，首先我们需要明白，点击了确定按钮后应该实现一些什么样的动作：需要将 **SpaceFrame** 变得不可见，显示 **EditorFrame**，并将用户所选择的工作空间目录的绝对路径传给 **EditorFrame**，告诉它选择了的目录，那么 **EditorFrame** 将会以这个目录作为工作的目录。

#### 8.4.2 实现工作空间选择的确定按钮

当选择完工作空间后，需要告诉 **EditorFrame** 选择了哪个目录，再根据让 **EditorFrame** 根据选择的目录去创建项目树。在 **EditorFrame** 类中，创建界面的是 **initFrame** 方法，因此，只需要为 **initFrame** 方法传入一个对象即可。先创建一个工作目录的对象：**WorkSpace** 类，该类需要保存工作空间的目录 **File** 对象。

代码清单：code\editor\src\org\crazyit\editor\commons\WorkSpace.java

```
public class WorkSpace {
    private File folder; //工作空间对应的目录
    public WorkSpace(File folder) {
        this.folder = folder;
    }
    ...
}
```

下面，再为工作空间选择的确定按钮添加一个监听器，**ConfirmButtonListener** 类，用于处理用户点击了确定按钮的行为。

代码清单：code\editor\src\org\crazyit\editor\SpaceFrame.java

```
class ConfirmButtonListener implements ActionListener {
    private SpaceFrame spaceFrame;
    private EditorFrame editorFrame;
    public ConfirmButtonListener(SpaceFrame spaceFrame, EditorFrame editorFrame) {
        this.spaceFrame = spaceFrame;
        this.editorFrame = editorFrame;
    }
    public void actionPerformed(ActionEvent arg0) {
        //调 EditorFrame 的 initFrame 方法初始化界面
        editorFrame.initFrame(new WorkSpace(spaceFrame.getFolder()));
        editorFrame.setVisible(true); //将 EditorFrame 设为可见
        spaceFrame.setVisible(false); //让工作选择空间界面不可见
    }
}
```

这里我们需要创建一个 **WorkSpace** 对象，将用户选择工作目录与 **editorFrame** 作为构造参数，在 **EditorFrame** 调用 **initFrame** 建立界面的时候，就将这个 **WorkSpace** 对象作为方法参数传入（上面代码



的黑体部分), 并需要在 `EditorFrame` 类中提供一个 `WorkSpace` 的属性用于保存这个属性, 代码如下:  
 代码清单: `code\editor\src\org\crazyit\editor\EditorFrame.java`

```
//该编辑区所对应的工作空间
private WorkSpace workSpace;

...

public void initFrame(WorkSpace workSpace) {
    this.workSpace = workSpace;
    ...
}
```

为 `SpaceFrame` 的确定按钮添加监听器:

```
//为确定按钮添加监听器
confirmButton.addActionListener(new ConfirmButtonListener(this, editorFrame));
```

现在运行 `Main` 类, 可以看到当我们选择了工作空间目录, 并点击了确定后, 我们的编辑主界面就出来了。

注: 需要为 `SpaceFrame` 添加关闭事件, 当用户点击了取消或者关闭窗口时, 需要退出程序。

## 8.5 创建项目树

实现了工作空间的选择按钮后, 现在的 `EditorFrame` 就可以得到一个 `WorkSpace` 的对象, 该对象里面有工作空间的目录及对应的 `EditorFrame` 对象, 现在就根据这个 `WorkSpace` 对象来创建项目显示区的项目树。创建树, 我们可以使用 `swing` 中的 `JTree` 对象, 在创建项目显示区时, 我们已经定义了一棵示例的树暂时放在该区域。

### 8.5.1 创建树的各个对象

在创建项目树前, 我们需要将树的各个对象创建, 例如树所使用的 `model`, 树节点对象, 选择树节点时的监听器对象等。

首先, 先创建节点对象, 在这里, 我们将其命名为 `ProjectTreeNode`, 代码如下。

代码清单: `code\editor\src\org\crazyit\editor\tree\ProjectTreeNode.java`

```
public class ProjectTreeNode extends DefaultMutableTreeNode {
    private File file; //该节点对应的文件
    private List<ProjectTreeNode> children; //该节点下的子节点
    //ProjectTreeNode 的构造器, 参数分别时该节点对应的文件, 是否允许有子节点
    public ProjectTreeNode(File file, boolean allowsChildren) {
        super(file.getName(), allowsChildren);
        this.file = file;
        children = new ArrayList<ProjectTreeNode>(); //初始化该节点下的子节点集合
    }
    ...
}
```

`ProjectTreeNode` 对象里有该节点所对应的文件属性 (也可以是目录), 该节点的子节点集合, 每创建这样的一个对象, 表示就创建一个树的节点, 该对象继承于 `swing` 的 `DefaultMutableTreeNode` 类。创建完节点对象后, 我们还需要一个树的模型对象, 并继承于 `DefaultTreeModel` 类, 暂时只提供一个构造器, 用于设置这个模型的根节点, 代码如下。

代码清单: `code\editor\src\org\crazyit\editor\tree\ProjectTreeModel.java`

```
public class ProjectTreeModel extends DefaultTreeModel {
    public ProjectTreeModel(ProjectTreeNode arg0) {
```

```

        super(arg0);
    }
}

```

该类并没有任何覆盖父类的或者自己的行为，如果在下面需要加入自己的行为的时候，就可以在这个模型对象里面添加代码。现在还需要创建树的监听器，当用户点击了树的时候，那么就需要触发事件，触发的事件在本例中只是普通的打开文件。创建 `ProjectTreeSelectionListener` 类并继承 `MouseAdapter` 类。

代码清单：code\editor\src\org\crazyit\editor\tree\ProjectTreeSelectionListener.java

```

public class ProjectTreeSelectionListener extends MouseAdapter {
    public ProjectTreeSelectionListener() {
    }
    public void mousePressed(MouseEvent e) {
        //暂时提供空实现
    }
}

```

创建完上面的三个对象后，我们就可以在 `EditorFrame` 类中的 `initFrame` 创建项目树了，`EditorFrame` 的方法中只负责创建主编辑界面，如果将创建项目树的代码放到其中，好像有点不合适，这时，我们可以写一个 `TreeCreator` 的接口，传到 `EditorFrame` 中，再通过 `TreeCreator` 的接口方法去创建项目树，这样可以减低代码的耦合，`EditorFrame` 根本不需要理会 `TreeCreator` 是如何去创建树的。新建 `TreeCreator` 接口，并新增一个实现类 `TreeCreatorImpl`，我们需要明白这个 `TreeCreator` 提供些什么样的接口方法，例如需要一个根据一个 `WorkSpace` 对象去创建一个 `JTree` 的方法，还需要一个根据目录创建节点（`ProjectTreeNode`）的接口方法。

代码清单：code\editor\src\org\crazyit\editor\tree\TreeCreator.java

```

public interface TreeCreator {
    JTree createTree(EditorFrame editorFrame);根据编辑的 EditorFrame 对象创建项目树
    ProjectTreeNode createNode(File folder);根据一个目录创建它的节点
}

```

`TreeCeatorImpl` 暂时只提供空实现，接下来为 `EditorFrame` 的构造器添加构造参数，让 `EditorFrame` 在创建时就将 `TreeCreator` 作为构造参数传入，并为 `TreeCreator` 提供 `getter` 方法，再修改在 `Main` 类中创建 `EditorFrame` 对象的代码。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```

//负责创建树的对象
private TreeCreator treeCreator;
public EditorFrame(String title, TreeCreator treeCreator) {
    super(title); //设置标题
    this.treeCreator = treeCreator;
}
...

```

代码清单：code\editor\src\org\crazyit\editor\Main.java

```

public static void main(String[] args) {
    TreeCreator treeCreator = new TreeCreatorImpl();//创建一个 TreeCreator
    //创建 EditorFrame，暂时不用设置可见
    EditorFrame editorFrame = new EditorFrame("ide", treeCreator);
    //将 editorFrame 对象作为 SpaceFrame 的构造参数
    SpaceFrame spaceFrame = new SpaceFrame(editorFrame);
    spaceFrame.setVisible(true); //让 SpaceFrame 可见
}

```

以上代码的黑体部分创建一个 `TreeCreator`。那么，现在就可以去修改 `EditorFrame` 中的代码，使它用 `TreeCreator` 来创建项目树。

代码清单: code\editor\src\org\crazyit\editor\EditorFrame.java

```
public void initFrame(WorkSpace workSpace) {  
    ...  
    //使用 treeCreator 创建树  
    tree = treeCreator.createTree(this);  
    ...  
}
```

### 8.5.2 实现创建项目树的功能

现在我们就需要去考虑如何在 `TreeCreatorImpl` 的 `createTree` 方法中去创建一棵项目树了。我们得到工作空间的目录，这样，再通过一定的规则查找工作空间下的项目目录。本例使用的规则为查找工作空间下以 `.project` 结尾的文件，如果查找到一份，就确定该工作空间有一个项目，再通过字符串截取获得项目名称。例如，工作空间下面有一份“`project1.project`”的文件，那么就意味着有一个叫 `project1` 的项目存在于这个工作空间下。规则已经制定，那么现在就可以去实现 `createTree` 方法了。

代码清单: code\editor\src\org\crazyit\editor\tree\TreeCreatorImpl.java

```
public JTree createTree(WorkSpace workSpace) {  
    File spaceFolder = workSpace.getFolder();//获取 workSpace 中的工作空间目录  
    //创建树的根节点，类型为 ProjectTreeNode，根节点对应的文件为工作空间目录  
    ProjectTreeNode root = new ProjectTreeNode(spaceFolder, true);  
    ProjectTreeModel treeModel = new ProjectTreeModel(root); //以根节点建立一个树模型对象  
    return new JTree(treeModel);  
}
```

以上代码创建树的根节点对象和树的模型对象，并用 `treeModel` 对象创建 `JTree` 并返回。得到的树的效果如图 8.11。

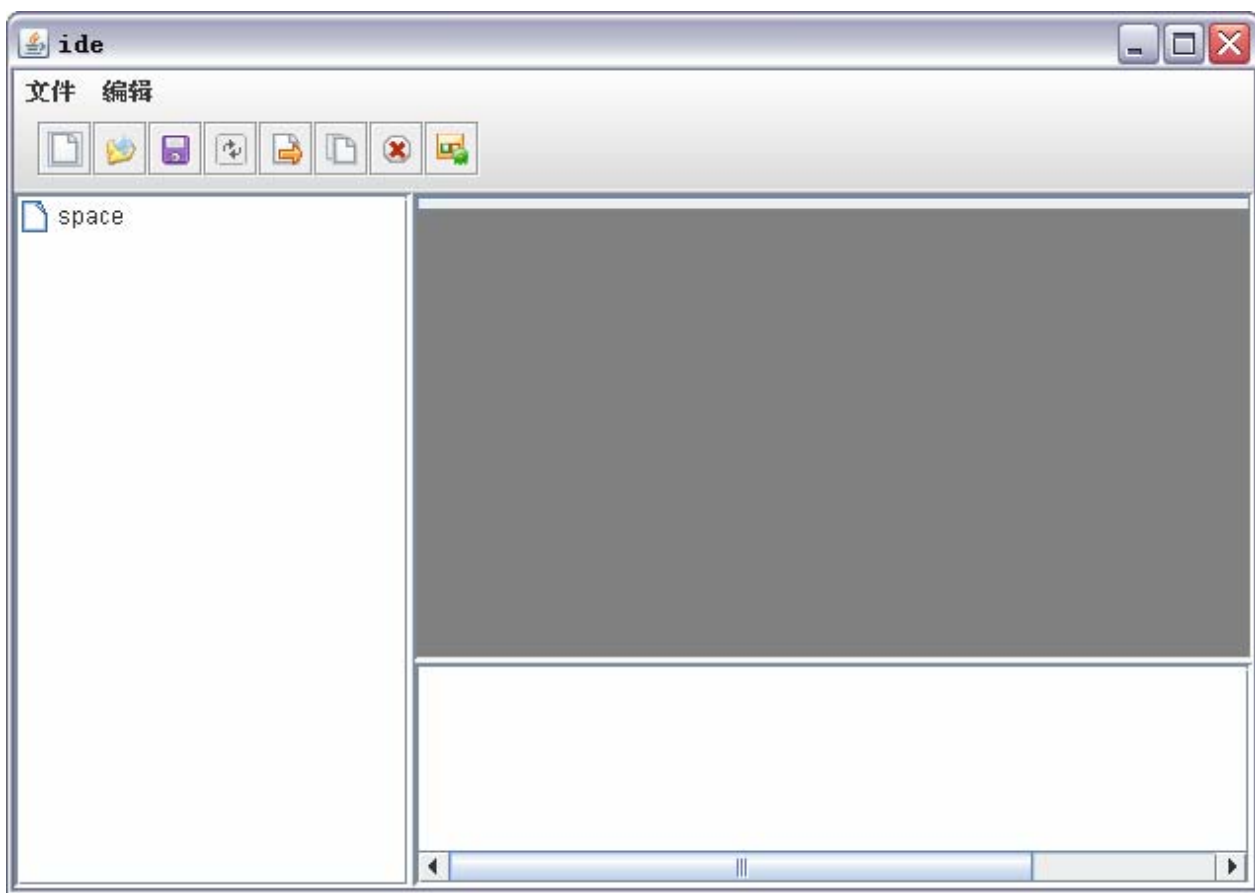


图 8.11 创建只有根节点的树

创建完这两个对象后，我们需要写一些私有方法，用于获取工作空间目录下的项目目录的集合，而要得到这些文件目录，我们需要写两个私有方法。

代码清单：code\editor\src\org\crazyit\editor\tree\TreeCreatorImpl.java

```
//获取工作空间目录下所有的项目名称
private List<String> getProjectNames(File spaceFolder) {
    List<String> result = new ArrayList<String>();
    //遍历工作空间下的所有文件
    for (File file : spaceFolder.listFiles()) {
        //获取以.project 结尾的文件
        if (file.getName().endsWith(".project")) {
            //添加到结果集中
            result.add(file.getName().substring(0, file.getName().indexOf(".project")));
        }
    }
    return result;
}
```

上面的方法获取工作空间目录下所有以.project 结尾的文件，再获取这些文件的文件名（不要后缀）。

代码清单：code\editor\src\org\crazyit\editor\tree\TreeCreatorImpl.java

```
//获取工作空间目录下所有的项目目录
private List<File> getProjectFolders(File spaceFolder) {
    //调用获取所有项目名称的私有方法
    List<String> projectNames = getProjectNames(spaceFolder);
}
```

```

List<File> result = new ArrayList<File>();
//获取工作空间下面所有的文件
File[] files = spaceFolder.listFiles();
for (String projectName : projectNames) {
    for (File file : files) {
        //如果工作空间下面的文件是目录，再去判断是否是项目目录
        if (file.isDirectory()) {
            if (projectName.equals(file.getName()))result.add(file);
        }
    }
}
return result;
}

```

注意以上代码的黑体部分，两个 **if** 判断，在遍历工作空间下的所有文件时，先判断该文件是否为目录，如果是目录，则再去判断该目录的名称是否为已有的项目名，可能会有这样的情况：例如工作空间下有一个 **test.project** 的文件，但是却没有相对应的项目目录，因此上面的程序将会忽略这一个项目。

现在可以去创建节点的对象，树中的每一个节点都对应一个 **ProjectTreeNode** 对象，每一个 **ProjectTreeNode** 对象下面都可能有的子节点，之所以说是可能，是因为在我们当前的情况下，该节点（**ProjectTreeNode**）的 **file** 属性是目录的时候，该节点都允许拥有子节点，当 **file** 属性是普通的文件时，就不允许拥有子节点了。编写创建直接节点的代码，也就是根据一个目录，创建它的子节点（直接子节点）。

代码清单：code\editor\src\org\crazyit\editor\tree\TreeCreatorImpl.java

```

//根据一个目录创建它的所有直接节点
private List<ProjectTreeNode> createNodes(File folder) {
    File[] files = folder.listFiles();//获取该目录下的所有文件
    List<ProjectTreeNode> result = new ArrayList<ProjectTreeNode>();
    //对该目录下的所有文件的数组进行两次遍历
    for (File file : files) {
        //第一次遍历，如果是目录的话，就加入到结果集合中
        if (file.isDirectory()) result.add(new ProjectTreeNode(file, true));
    }
    for (File file : files) {
        //第二次遍历，如果非目录的话，就加入到结果集合中
        if (!file.isDirectory()) result.add(new ProjectTreeNode(file, false));
    }
    return result;
}

```

注：以上的代码，之所以要对文件数组遍历两次，是因为按照一般的习惯，目录一般在前面显示，而一般的文件会在目录的后面显示，这里无形中给目录中的所有文件（包括子目录）进行了简单的排序。

接下来，再去实现 **TreeCreator** 的另外一个接口方法，虽然这时候 **TreeCreatorImpl** 中的 **createTree** 方法尚未实现，我们可以先去实现 **createNode(File folder)** 方法，这是因为在 **TreeCreatorImpl** 中将会使用这个方法去创建树。

代码清单：code\editor\src\org\crazyit\editor\tree\TreeCreatorImpl.java

```

//根据一个目录去创建该目录所对应的节点对象，该对象的所有的子节点都已经创建
public ProjectTreeNode createNode(File folder) {
    //创建一个父节点，即本方法即将返回的节点对象
    ProjectTreeNode parent = null;
    //如果参数 folder 不是一个目录的话，创建一个 ProjectTreeNode 对象并返回，表明它不允许拥有子节点
    if (!folder.isDirectory()) {

```

```

        return new ProjectTreeNode(folder, false);
    } else {
        //如果是一个目录的话，则创建上面的 parent，表明它是一个目录，可以拥有子节点
        parent = new ProjectTreeNode(folder, true);
    }
    //利用上面的 parent 节点去查找它下面所有的直接节点
    List<ProjectTreeNode> nodes = createNodes(parent.getFile());
    //获取到 parent 下面的所有直接子节点后，再去循环递归调用本方法
    for (ProjectTreeNode node : nodes) {
        //递归创建子节点，并将返回的节点添加到 parent 中
        parent.add(createNode(node.getFile()));
    }
    return parent;
}
}

```

写完上面的方法，就万事俱备，可以去实现创建树的方法了，前面该方法只实现了一半，创建了树的根，现在将其补充完整，让它成为一棵真正的树。

代码清单：code\editor\src\org\crazyit\editor\tree\TreeCreatorImpl.java

```

public JTree createTree(EditorFrame editorFrame) {
    //获取 workSpace 中的工作空间目录
    File spaceFolder = editorFrame.getWorkSpace().getFolder();
    //创建树的根节点，类型为 ProjectTreeNode，根节点对应的文件为工作空间目录
    ProjectTreeNode root = new ProjectTreeNode(spaceFolder, true);
    ProjectTreeModel treeModel = new ProjectTreeModel(root); //以根节点建立一个树模型对象
    JTree tree = new JTree(treeModel); //创建树对象
    //获取工作空间下面所有的目录（即与有 projectName.project 相对应的目录），也就是项目目录
    List<File> projectFolders = getProjectFolders(spaceFolder);
    //遍历项目目录集合，并为其创建子节点
    for (int i = 0; i < projectFolders.size(); i++) {
        //获取循环中的目录
        File projectFolder = projectFolders.get(i);
        //调用 createNode 创建它所有的子节点
        ProjectTreeNode node = createNode(projectFolder);
        //向根节点添加子节点（项目目录）
        root.add(node);
    }
    return tree;
}
}

```

这样实现了创建树的功能了，我们可以往所选择的工作空间中添加几个文件目录和文件加以测试。可以看到的效果如图 8.12 所示。



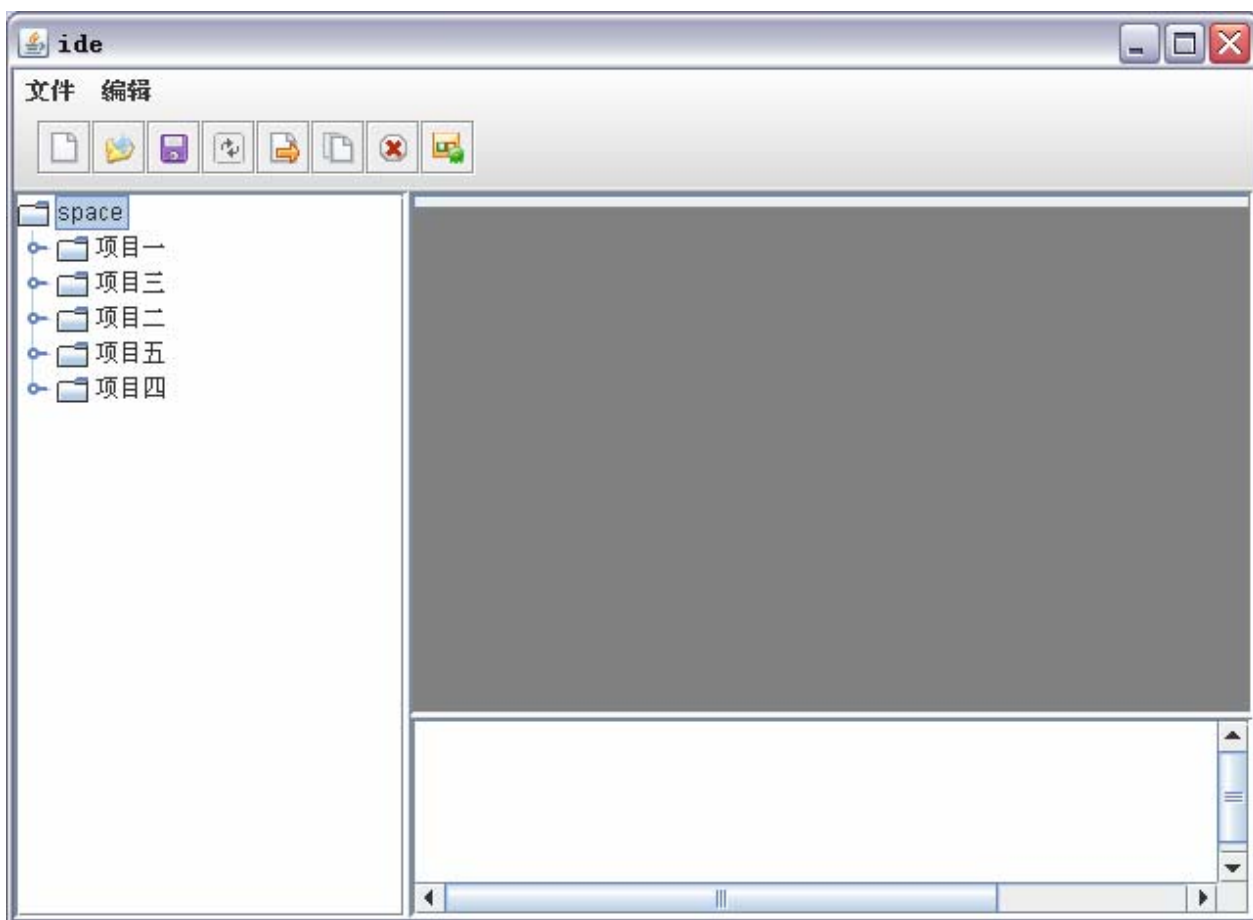


图 8.12 创建一棵完整的树

我们可以看到这棵树，以工作空间所在的目录作为根节点，整棵树创建完了。工作空中的几个项目都可以显示在树上了。

### 8.5.3 设置树的外观

在一般的工作空间下，只会显示各个项目，因此，可以将这棵树的根节点隐藏，使用 `JTree` 的 `setRootVisible` 方法即可。

代码清单：code\editor\src\org\crazyit\editor\tree\TreeCreatorImpl.java

```
public JTree createTree(WorkSpace workSpace) {  
    ...  
    tree.setRootVisible(false); //设置树的根节点不可见  
    return tree;  
}
```

除了不显示树的根之外，我们还可以定制树的各种图片，使用 `DefaultTreeCellRenderer` 来定制树的图片。先编写一个图片文件的读取类 `ImageUtil`，用于读取树所用到的图片：

代码清单：code\editor\src\org\crazyit\editor\util\ImageUtil.java

```
public class ImageUtil {  
    public static String FOLDER_CLOSE = "images/folder-close.gif"; //目录关闭图片  
    public static String FOLDER_OPEN = "images/folder-open.gif"; //目录打开图片  
    public static String FILE = "images/file.gif"; //无子节点的文件图片
```

```
//使用 ImageIO 流读取图片
public static Image getImage(String path) {
    return ImageIO.read(new File(path));
}
public static ImageIcon getImagelcon(String path) {
    return new ImageIcon(getImage(path));
}
}
```

代码清单：code\editor\src\org\crazyit\editor\tree\TreeCreatorImpl.java

```
public JTree createTree(WorkSpace workspace) {
    ...
    DefaultTreeCellRenderer renderer = new DefaultTreeCellRenderer();//定制节点图片
    renderer.setOpenIcon(ImageUtil.getImagelcon(ImageUtil.FOLDER_OPEN)); //目录打开时的图片
    renderer.setLeafIcon(ImageUtil.getImagelcon(ImageUtil.FILE)); //节点没有子节点的图片
    renderer.setClosedIcon(ImageUtil.getImagelcon(ImageUtil.FOLDER_CLOSE)); //目录关闭时的图片
    tree.setCellRenderer(renderer); //设置树的部件处理类为上面的 renderer
    tree.setRootVisible(false); //设置树的根节点不可见
    return tree;
}
```

这时候再去运行 **Main** 类，可以发现在项目显示区并没有树，这是由于我们隐藏了根，但是根却没有自动展开，我们可以获取树的根，再对其进行展开（创建的时候）。

代码清单：code\editor\src\org\crazyit\editor\tree\TreeCreatorImpl.java

```
public JTree createTree(WorkSpace workspace) {
    ...
    TreePath path = new TreePath(root); //创建根在项目树中的路径
    tree.expandPath(path); //让树默认展开根节点
    tree.setRootVisible(false); //设置树的根节点不可见
    return tree;
}
```

再次运行 **Main** 类，并选择工作空间，可以看到如图 8.13 的效果。

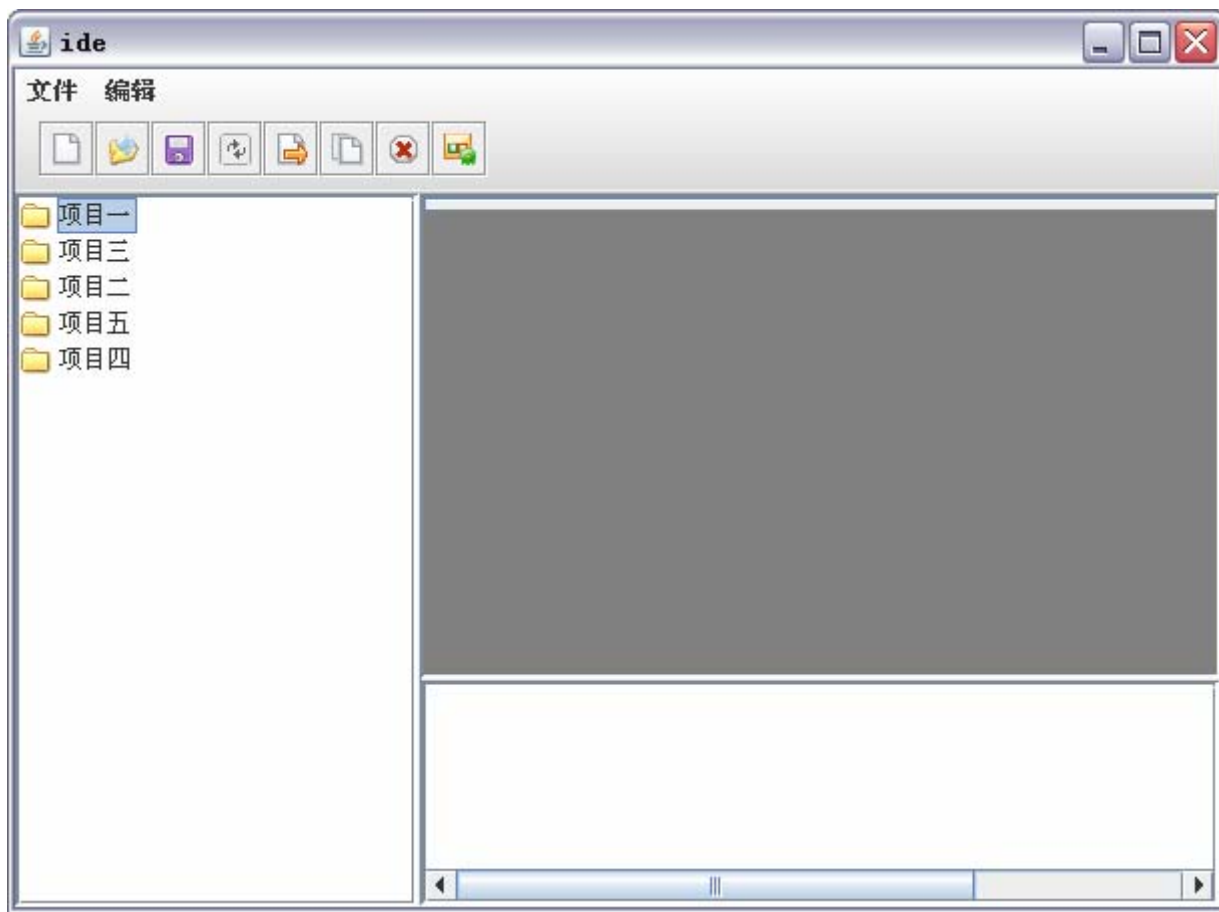


图 8.13 设置树的外观

#### 8.5.4 实现树节点的选择事件

在 8.5.1 中，已经创建了树的各个相关的类，其中就包括树的节点的点击监听类，当时创建该类的时候并没有对它进行实现，现在我们的树已经成型了，可以对加入节点的选择事件。首先我们需要明白点击节点时需要做些什么样的事情，当点击节点的时候，如果点击的是一个目录，那么就展开这个目录，这个 `JTree` 已经帮我们实现了，如果点击的是一个普通节点（普通文件），那么就将该节点的文件内容显示到 `EditorFrame` 中的主编辑区的文本编辑域中。现在我们先开发一个读取文件内容的工具类，在点树节点的时候读取节点（文件）的内容，这样在监听器的类，就不需要关心如何去读取文件了。新建一个 `FileUtil` 的类。

代码清单：code\editor\src\org\crazyit\editor\util\FileUtil.java

```
//读取一个文件的内容
public static String readFile(File file) {
    StringBuffer result = new StringBuffer();
    try {
        FileInputStream fis= new FileInputStream(file); //使用 IO 流读取文件
        String content = null;
        byte[] arr = new byte[1024];
        int readLength ;
        while ((readLength = fis.read(arr)) > 0) {
            content = new String(arr, 0, readLength);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return result.append(content).toString();
}
```

```

        result.append(content);
    }
    fis.close();
} catch(IOException e) {
    throw new FileNotFoundException("read " + file.getAbsolutePath() + " file error");
}
return result.toString();
}

```

注: `FileNotFoundException` 是自定义异常类, 继承 `RuntimeException`。

以上代码创建一个读取文件的方法, 最终的返回值为该文件的内容字符串。现在可以去实现树的监听器类 `ProjectTreeSelectionListener` 的 `mousePressed` 方法, 当鼠标在树上点了的时候, 就去判断是否点击了文件节点, 如果点击了文件节点, 则调 `FileUtil` 的读取文件的方法获取文件内容。

```

private EditorFrame editorFrame; //主界面的 frame
//将主界面的 frame(EditorFrame)作为构造参数传入监听器
public ProjectTreeSelectionListener(EditorFrame editorFrame) {
    this.editorFrame = editorFrame;
}

```

代码清单: `code\editor\src\org\crazyit\editor\tree\TreeCreatorImpl.java`

```

public JTree createTree(EditorFrame editorFrame) {
    ...
    //为项目树添加一个树选择监听器
    tree.addMouseListener(new ProjectTreeSelectionListener(editorFrame));
    ...
    return tree;
}

```

将 `EditorFrame` 对象传给树的监听器后, 监听器就可以调用 `EditorFrame` 的方法去处理 `EditorFrame` 对象中的属性, 例如可以调用 `EditorFrame` 中的打开文件方法。在 `EditorFrame` 中新建一个获取当前树节点的方法, 树监听器得到当前所选择的节点, 就可以对其进行处理。

代码清单: `code\editor\src\org\crazyit\editor\EditorFrame.java`

```

//获取编辑器主界面项目树中所选中的节点
public ProjectTreeNode getSelectNode() {
    //获得当前树选择的节点在树中的路径
    TreePath path = tree.getSelectionPath();
    //如果当前选择了节点
    if (path != null) {
        //创建一个 ProjectTreeNode 对象并用于返回
        ProjectTreeNode selectNode = (ProjectTreeNode)path.getLastPathComponent();
        return selectNode;
    }
    //当前没有选择节点就返回 null
    return null;
}

```

那么监听器就可以使用这个方法得到当前树所选择的节点。

代码清单: `code\editor\src\org\crazyit\editor\tree\ProjectTreeSelectionListener.java`

```

public void mousePressed(MouseEvent e) {
    //得到当前所选择的节点
    ProjectTreeNode selectNode = this.editorFrame.getSelectNode();
    //如果没有选择节点, 就返回
    if (selectNode == null) return;
}

```

```

//如果选择的是一个目录，返回
if (selectNode.getFile().isDirectory()) return;
//使用 EditorFrame 的方法来打开文件
this.editorFrame.openFile(selectNode.getFile());
}

```

注：以上代码的黑体部分，在监听器中，得到选中的节点后，就调用 EditorFrame 的 openFile 方法，这时需要在 EditorFrame 中新建一个 openFile 方法，参数是 File 对象，暂时提供空实现，需要处理打开文件的操作。

## 8.6 实现文件操作功能

这一节，我们将实现文件的操作功能，包括新建文件、目录和项目，文件的打开、保存、运行、刷新等一系列基本的文件操作。

### 8.6.1 新建文件

我们在 8.3.3 一节中创建了文件操作的菜单和工具栏，在 8.3.5 一节中建立了添加的界面，现在我们来实现新建文件的功能。要新建一个文件，前提是需要用户在树节点上选了一个节点，得到创建文件的目录。在 8.3.5 一节中提到，新建文件、目录和项目是共用一个添加界面的，下面我们先新建一个 AddHandler 的接口，用于处理新建的操作。为这个接口新建一个实现类 AddFileHandler，用于处理新建文件的操作。

代码清单：code\editor\src\org\crazyit\editor\handler\add\AddHandler.java

```

public interface AddHandler {
    //新增完后需要做的事情，需要做的事情由实现类去实现
    //参数为 EditorFrame，AddFrame，输入的信息 data
    void afterAdd(EditorFrame editorFrame, AddFrame addFrame, Object data);
}

```

代码清单：code\editor\src\org\crazyit\editor\handler\add\AddFileHandler.java

```

public void afterAdd(EditorFrame editorFrame, AddFrame addFrame, Object data) {
    //输入完文件名称后，就执行这里的代码
}

```

在 EditorFrame 打开添加的界面（AddFrame）的时候，我们需要告诉 AddFrame 一些相关的信息，例如要使用哪个添加处理类，所显示的一些文字等，我们新建一个 AddInfo 的类，用于保存这些信息。

代码清单：code\editor\src\org\crazyit\editor\commons\AddInfo.java

```

public class AddInfo {
    private String info; //字符串，在新增界面的 text 前显示，例如：文件名称
    private EditorFrame editorFrame; //受新增操作影响的 frame
    private AddHandler handler; //新增完后的处理类
    public AddInfo(String info, EditorFrame editorFrame, AddHandler handler) {
        this.info = info;
        this.editorFrame = editorFrame;
        this.handler = handler;
    }
    ...
}

```

注意上面的黑体部分，表示让添加界面所显示的文字，需要显示的文字请看图 8.14。



图 8.14 AddInfo 说明

注：图 8.14 中的“需要显示文字的地方”就是 AddInfo 中的 info 属性。

现在，回到 EditorFrame 中，为新建文件添加监听器，并封装一个 AddInfo 传给 AddFrame。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```
//添加的界面
private AddFrame addFrame;
//新建文件的 Action 对象
private Action fileNew = new AbstractAction("新建文件", new ImageIcon("images/newFile.gif")) {
    public void actionPerformed(ActionEvent e) {
        newFile();
    }
};
//新建文件的方法
public void newFile() {
    AddInfo info = new AddInfo("文件名称: ", this, new AddFileHandler());
    showAddFrame(info);
}
//显示新增的界面
private void showAddFrame(AddInfo info) {
    //使 EditorFrame 变为不可用
    setEnabled(false);
    addFrame = new AddFrame(info);
    addFrame.pack();
    addFrame.setVisible(true);
}
```

注：需要修改 AddFrame 的构造器，需要一个 AddInfo 对象作为构造参数。

上面代码中新增了两个方法，实现了新建文件的菜单，显示添加的界面，并将 AddInfo 对象传到 AddFrame 中，这样 AddFrame 就知道如何显示对应的文字，添加完后就可以调用 AddInfo 中的 AddHandler 类去处理添加动作，至于怎样处理，AddFrame 完全不用关心。只负责调用 AddHandler 的方法即可。我们需要明确 AddFrame 所要做的事情：

- ☐ 文本框值改变的监听器。
- ☐ 取消按钮的监听器
- ☐ 窗口关闭的监听器
- ☐ 确定按钮的监听器

文本框值改变的监听器，与 8.4 中实现工作空间选择的功能一样，文本框有值的时候，确定按钮才需要显示。在文本框输入了值后，就可以调用 AddHandler 的方法处理，将 AddFrame 和 EditorFrame 对象传给处理类，剩下就是处理类的事情了。

代码清单：code\editor\src\org\crazyit\editor\AddFrame.java

```
String data = nameText.getText();//获取用户输入
//新增后会涉及的一些与业务相关的操作留给 Handler 类处理
info.getHandler().afterAdd(info.getEditorFrame(), this, data);
```

上面的代码中，通过 AddInfo 获得处理类，再调用 afterAdd 方法，将相关的信息传递给具体的处理



类，这里的处理类是 `AddFileHandler`。

代码清单：code\editor\src\org\crazyit\editor\handler\add\AddFileHandler.java

```
public void afterAdd(EditorFrame editorFrame, AddFrame addFrame, Object data) {
    ProjectTreeNode selectNode = editorFrame.getSelectedNode();//获得当前所选择的树节点
    File folder = selectNode.getFile();//获取当前选择节点所对应的文件
    //如果 folder 不是一个目录，则用 selectNode 的父节点（是一个目录）作为新文件的存放目录
    if (!folder.isDirectory()) {
        ProjectTreeNode parent = (ProjectTreeNode)selectNode.getParent();
        selectNode = parent;
        folder = parent.getFile();
    }
    //创建文件，放到 folder 下
    File newFile = new File(folder.getAbsoluteFile() + File.separator + data);
    newFile.createNewFile();
    ProjectTreeModel model = (ProjectTreeModel)editorFrame.getTree().getModel();
    model.reload(selectNode); //重新加载所选择的节点
    editorFrame.setEnabled(true); //使主编辑 frame 可用
    addFrame.setVisible(false); //让添加的 frame 不可见
}
```

运行 `Main` 类，并进行新建文件的操作，可以发现在某个目录下新建了一个文件，该节点并没有刷新，因此，我们需要为 `ProjectTreeModel` 实现一个 `reload` 方法，参数是 `TreeNode`、`TreeCreator`，以便达到我们需要刷新某个节点的效果。

代码清单：code\editor\src\org\crazyit\editor\tree\ProjectTreeModel.java

```
public void reload(ProjectTreeNode node, TreeCreator creator) {
    ProjectTreeNode parent = (ProjectTreeNode)node.getParent();//获取 node 节点的父节点
    if (parent == null) return; //父节点为 null，返回，不需要 reload
    int index = parent.getIndex(node); //获取 node 节点在父节点的索引
    parent.remove(index); //先装 node 节点从 parent 中删除
    node = creator.createNode(node.getFile()); //再通过 TreeCreator 获取新的节点
    parent.insert(node, index); //添加到父节点中
    super.reload(node); //调用 DefaultTreeModel 的 reload 方法
}
```

注：需要在 `EditorFrame` 中提供返回树对象（`getTree`）的方法。

在 `ProjectTreeModel` 中重新实现了 `reload` 方法后，可以将 `reloadNode` 的功能放到 `EditorFrame` 中，直接让 `EditorFrame` 刷新它自己的树。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```
//刷新树节点
public void reloadNode(ProjectTreeNode selectNode) {
    if (selectNode == null) return;
    //刷新树的节点
    ProjectTreeModel model = (ProjectTreeModel)getTree().getModel();
    //重新加载所选择的节点
    model.reload(selectNode, treeCreator);
}
```

回头再去修改 `AddFileHandler` 的 `afterAdd` 方法，在 `afterAdd` 中调用了 `EditorFrame` 的 `reloadNode` 方法，修改为：

```
//重新加载所选择的节点
editorFrame.reloadNode(selectNode);
```

运行 `Main` 类，添加文件的功能已经实现了，效果如图 8.15 与 8.16 所示。

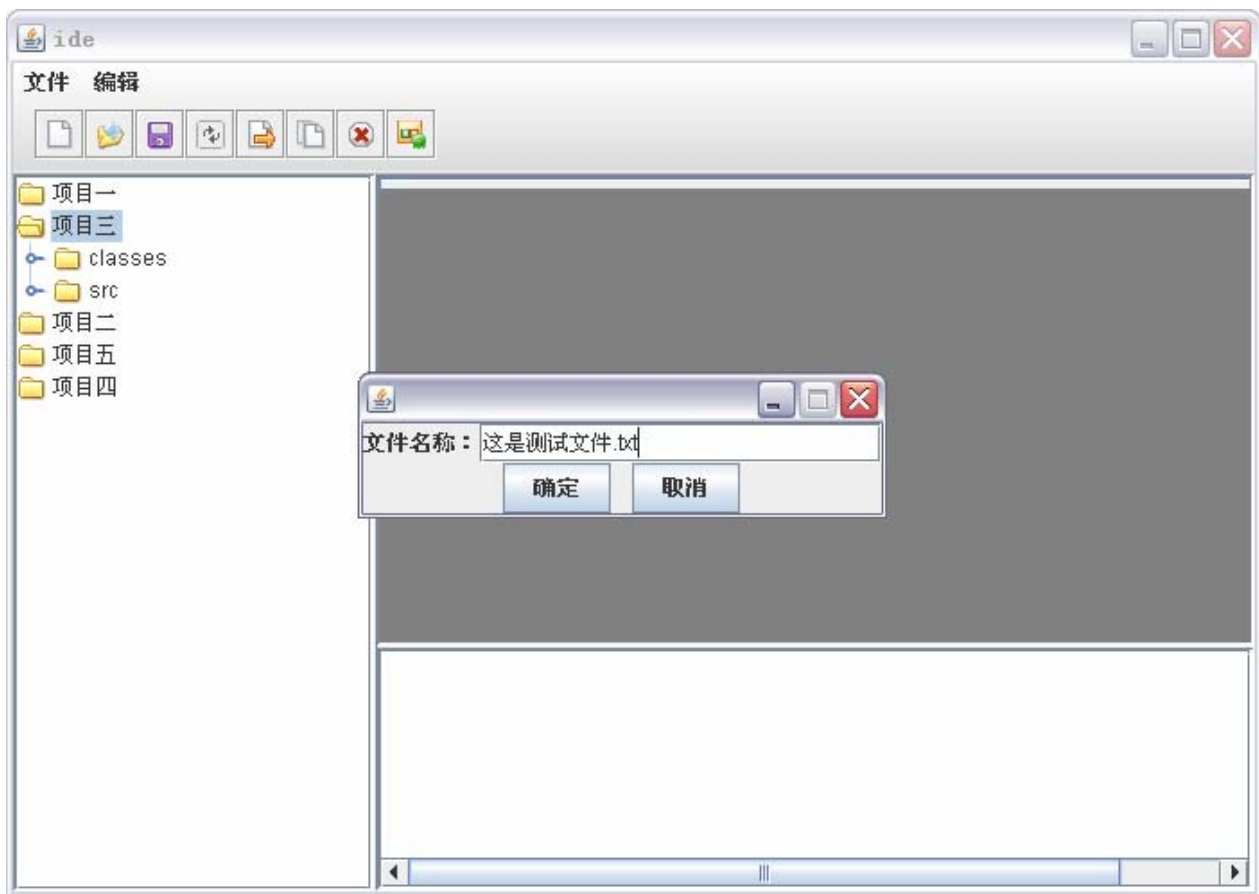


图 8.15 新建文件

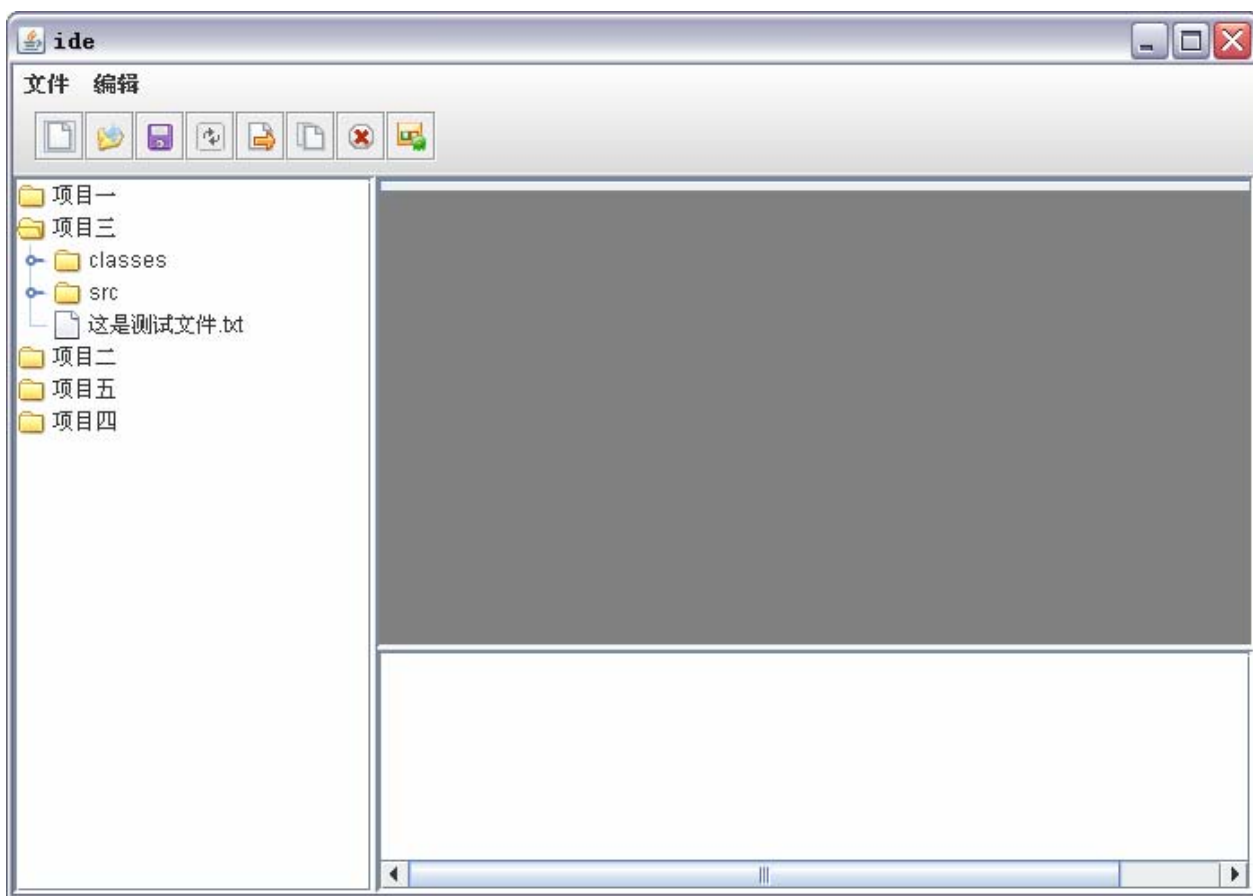


图 8.16 新建文件后刷新树

在上面的程序中，我们已经实现了文件操作中的刷新功能，只要为刷新菜单添加监听器，获取当前的节点，再进行用 `ProjectTreeModel` 的 `reload` 方法进行刷新树的某个节点即可。上面我们实现了树的刷新功能，那么现在可以实现 `EditorFrame` 中刷新的 `Action`。

代码清单：`code\editor\src\org\crazyit\editor\EditorFrame.java`

```
//刷新树的 Action 对象
private Action refresh = new AbstractAction("刷新", new ImageIcon("images/refresh.gif")) {
    public void actionPerformed(ActionEvent e) {
        reloadNode(getSelectNode());
    }
};
```

### 8.6.2 新建目录

在 8.6.1 中实现了新建文件的功能，在 `AddFrame` 中，我们只需要传给该类一个 `AddInfo` 对象，就可以根据这个对象中的处理类去新建文件、目录或者项目，这一节我们将在这个基础上实现新建目录的功能。只需要添加一个实现类 `AddFolderHandler` 就可以实现。

代码清单：`code\editor\src\org\crazyit\editor\handler\add\AddFolderHandler.java`

```
public void afterAdd(EditorFrame editorFrame, AddFrame addFrame, Object data) {
    ProjectTreeNode selectNode = editorFrame.getSelectNode();//获得树中所选取的节点
    File folder = selectNode.getFile();//获取该节点所对应的文件对象
    //如果 folder 不是一个目录，则用 selectNode 的父节点（是一个目录）作为新目录的父目录
```

```

        if (!folder.isDirectory()) {
            ProjectTreeNode parent = (ProjectTreeNode)selectNode.getParent();
            selectNode = parent; //让当前所选择的文件的父目录作为当前选择的目录
            folder = parent.getFile();
        }
        //创建一个文件目录对象
        File newFolder = new File(folder.getAbsoluteFile() + File.separator + data);
        newFolder.mkdir(); //创建新的目录
        editorFrame.reloadNode(selectNode); //刷新树的节点
        editorFrame.setEnabled(true); //让 EditorFrame 可用
        addFrame.setVisible(false); //让添加的 frame 不可见
    }
}

```

上面的代码是添加目录后的实现，同样的在所选节点下创建一个目录，创建完后再刷新父节点。下面代码实现了新建目录 **Action** 的方法。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```

//新建目录的 Action 对象
private Action folderNew = new AbstractAction("新建目录", new ImageIcon("images/newFile.gif")) {
    public void actionPerformed(ActionEvent e) {
        newFolder();
    }
};
//新建目录的方法
public void newFolder() {
    AddInfo info = new AddInfo("目录名称: ", this, new AddFolderHandler());
    showAddFrame(info);
}

```

注：上面代码中的黑体部分，使用了 **AddFolderHandler** 实现类。这样的效果就和新建文件的效果一样，只是新建的是一个目录。

### 8.6.3 新建项目

与 8.6.2 的新建目录功能一样，只需要加多一个 **AddHandler** 的实现类，并为菜单加入监听器即可。新建 **AddProjectHandler** 类，但 **AddProjectHandler** 与新建文件、目录的实现有所不同，不仅是新建一个项目目录，还需要新建一个 **name.project** 的项目对应文件，还需要新建一些与项目相关的目录，例如源文件目录（**src**），Java 文件的编译目录（**classes**）等。新建完项目后，还需要刷新整棵树（刷新工作空间），因此，我们在 **EditorFrame** 中提供一个刷新整棵树的方法，以便为这个实现类所调用。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```

//重新在 treePane 中构造一次树
public void refreshTree(JTree newTree) {
    //将 tree 对象变为参数中的 newTree
    this.tree = newTree;
    //让 treePane（放树的容器）设置 newTree 为新的视图
    treePane.setViewportViewView(newTree);
    //更新界面
    treePane.updateUI();
}

```

在新建目录的时候，还需要有源文件所在的目录，编译目录等信息。新建一个 **CompileConfig** 类来存放这些相关的信息。当然，如果做得更完善一点，可以提供界面来配置这些信息。下面将实现新建项目的处理类 **AddProjectHandler**。

代码清单: code\editor\src\org\crazyit\editor\handler\add\AddProjectHandler.java

```
public void afterAdd(EditorFrame editorFrame, AddFrame addFrame, Object data) {
    File spaceFolder = editorFrame.getWorkSpace().getFolder();//获取工作空间所在的目录
    //创建.project 文件
    File projectFile = new File(spaceFolder.getAbsolutePath() +
        File.separator + data + ".project");
    //创建项目目录
    File projectFolder = new File(spaceFolder.getAbsolutePath() + File.separator + data);
    //项目文件不存在, 创建项目文件
    if (!projectFile.exists()) projectFile.createNewFile();
    //项目目录不存在, 创建项目文件目录
    if (!projectFolder.exists()) projectFolder.mkdir();
    //创建项目的 src 目录和编译目录
    File src = new File(projectFolder.getAbsolutePath() +
        File.separator + CompileConfig.SRC_DIR);
    //Java 文件编译的输入目录
    File output = new File(projectFolder.getAbsolutePath() +
        File.separator + CompileConfig.OUTPUT_DIR);
    //创建 src 和 output 两个目录
    src.mkdir();
    output.mkdir();
    JTree newTree = editorFrame.getTreeCreator().createTree(editorFrame); //刷新整棵树
    editorFrame.refreshTree(newTree);
    editorFrame.setEnabled(true); //让 EditorFrame 变得可用
    addFrame.setVisible(false); //让当前所选择的文件的父目录作为当前选择的目录
}
```

上面代码中的黑体部分分别创建 **name.project** 文件、源文件存放目录和编译输出目录。新建完 **name.project**、源文件存放目录和编译目录后, 再调用 **EditorFrame** 中的刷新整棵树的方法去刷新项目树。接下来再为 **EditorFrame** 类中的新建项目菜单添加监听器。

代码清单: code\editor\src\org\crazyit\editor\EditorFrame.java

```
//新建项目的 Action 对象
private Action projectNew = new AbstractAction("新建项目", new ImageIcon("images/newFile.gif")) {
    public void actionPerformed(ActionEvent e) {
        newProject();
    }
};
//新建项目的方法
public void newProject() {
    AddInfo info = new AddInfo("项目名称: ", this, new AddProjectHandler());
    showAddFrame(info);
}
```

实现完上面的代码后, 我们去运行 **Main** 类, 即可看到效果, 如图 8.17 和图 8.18 所示。

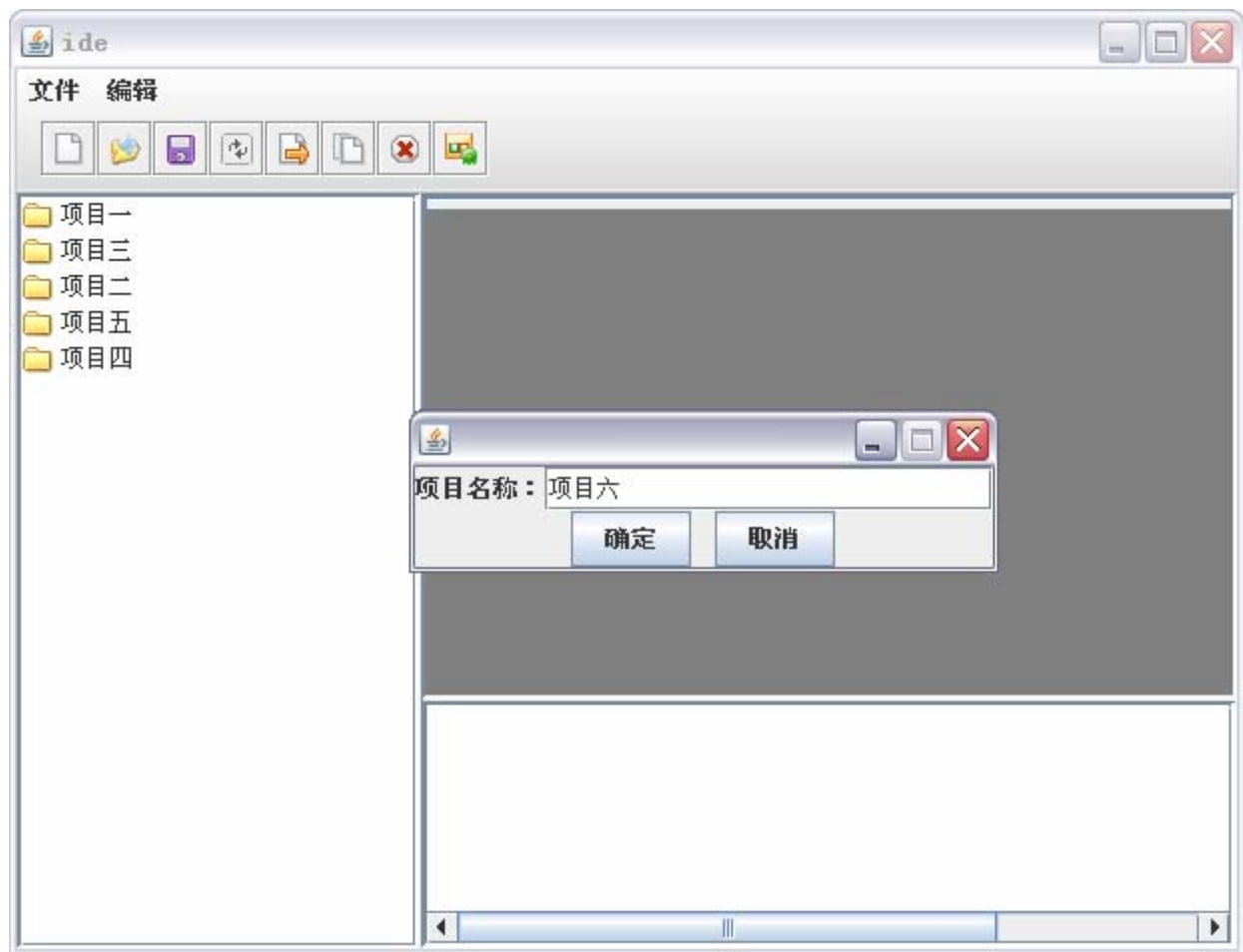


图 8.17 实现建新项目



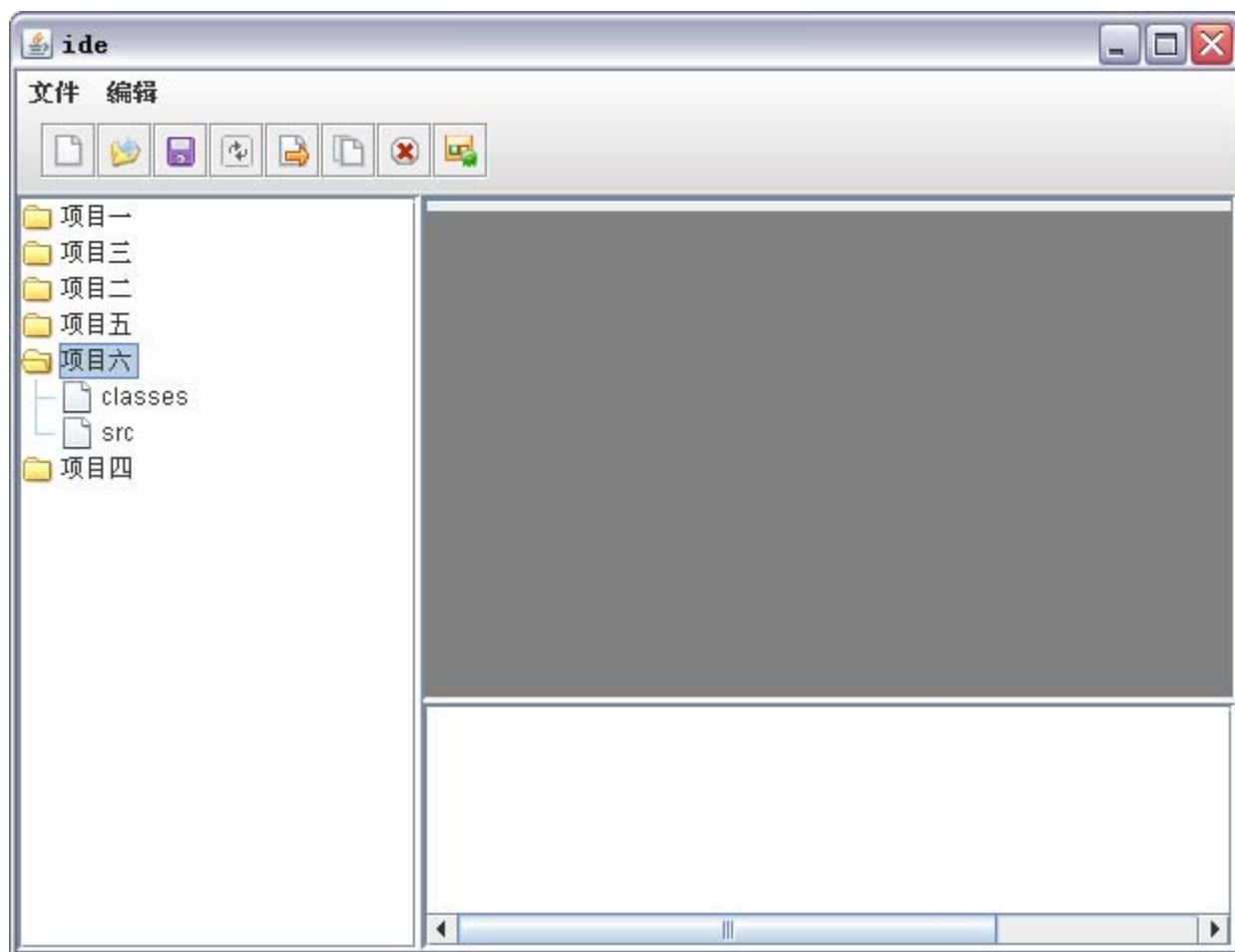


图 8.18 新建项目后刷新整棵树

创建完项目后，IDE 会帮我们创建 `src` 和 `classes` 目录，并创建 `name.project` 的文件，下次进入该工作空间后，该项目就可以在项目树中显示。

#### 8.6.4 策略模式

到此，我们已经实现了新建文件、目录和项目的功能，在这个过程中，我们不知不觉的使用 23 种设计模式中的策略模式。

策略模式的参与者主要有：

- ❑ **Strategy**: 定义算法的公共接口，让 **Context** 来调用某个 **ConcreteStrategy** 定义的算法。
- ❑ **ConcreteStrategy**: 以 **Strategy** 为接口实现某个具体的算法。
- ❑ **Context**: 使用一个 **Strategy** 进行配置，维护一个对 **Strategy** 对象的引用。

我们在 8.6.1 中新建文件时，我们就新建了一个 **AddHandler** 的接口，并对其进行具体不同的实现，它的实现类包括：**AddFileHandler**，**AddFolderHandler**，**AddProjectHandler**。在策略模式中的 **Context** 参与者，就是我们前面所写的 **AddFrame** 类，该类维护一个 **AddHandler** 接口，并在新增界面点击确定后调用该接口的 `afterAdd` 方法。在这里策略模式所体现的优势就是 **AddFrame** 不需要去理会 **AddHandler** 的具体实现，只需要在 **EditorFrame** 构造不同的 **AddHandler** 给 **AddFrame** 即可。

### 8.6.5 文件打开

在本章 8.5.4 中，已经实现了树节点的监听器，点击了树的某个节点的时候，就调用 `EditorFrame` 中的 `openFile` 方法打开一份文件，在 8.5.4 中并没有去实现 `openFile` 方法，在本小节中将会去实现这个方法，处理文件的打开，并显示在编辑区中。

首先我们需要明白，打开文件时，IDE 需要做的事情：

- ❑ 新建一个窗体对象，使用 `JInternalFrame` 类
- ❑ 新建一个文本编辑对象，我们这里使用 `JTextPane` 类，并将其放到 `JInternalFrame` 中
- ❑ 将新建的 `JInternalFrame` 对象放到 `EditorFrame` 中的 `JDesktopPane` 容器中
- ❑ 再设置对应的 `JTabbedPane` 对象

在新建文本编辑对象的时候，需要得到被打开的文件，再使用 IO 读取文件的内容。下面新建一个 `JTextPane` 的子类，表示一个文本的编辑域对象。该对象需要实现

代码清单：code\editor\src\org\crazyit\editor\EditPane.java

```
public class EditPane extends JTextPane {  
    public EditPane(File file){  
    }  
}
```

`EditPane` 提供一个构造器，参数为一个 `File` 对象，表示这个 `EditPane` 所对应的文件，也就是这个 `EditPane` 所需要显示的内容的来源文件。实现 `EditorFrame` 的 `openFile` 方法，新建窗体，新建文本编辑对象，新建 `JInternalFrame` 对象，最后设置 `JTabbedPane` 对象。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```
//打开文件的方法  
public void openFile(File file) {  
    //设置 EditorFrame 的标题为该文件的全路径  
    setTitle(file.getAbsolutePath());  
    //创建一个 JInternalFrame 对象，title 为文件的绝对路径  
    JInternalFrame iframe = new JInternalFrame(file.getAbsolutePath(), true, true, true, true);  
    //新建一个 EditPane 对象  
    EditPane editPane = new EditPane(file);  
    iframe.add(new JScrollPane(editPane));  
    desk.add(iframe);  
    iframe.show();  
    //设置 iframe 的位置与大小  
    iframe.reshape(0, 0, 400, 300);  
    //在 tabPane 中添加一个 tab 页，tab 页面的标题为文件名称，tab 的 tips 为文件的绝对路径  
    tabPane.addTab(file.getName(), null, null, file.getAbsolutePath());  
    tabPane.setSelectedIndex(tabPane.getTabCount() - 1);  
}
```

运行 `Main` 类，再点击某个文件节点，即可看到如图 8.19 的效果。

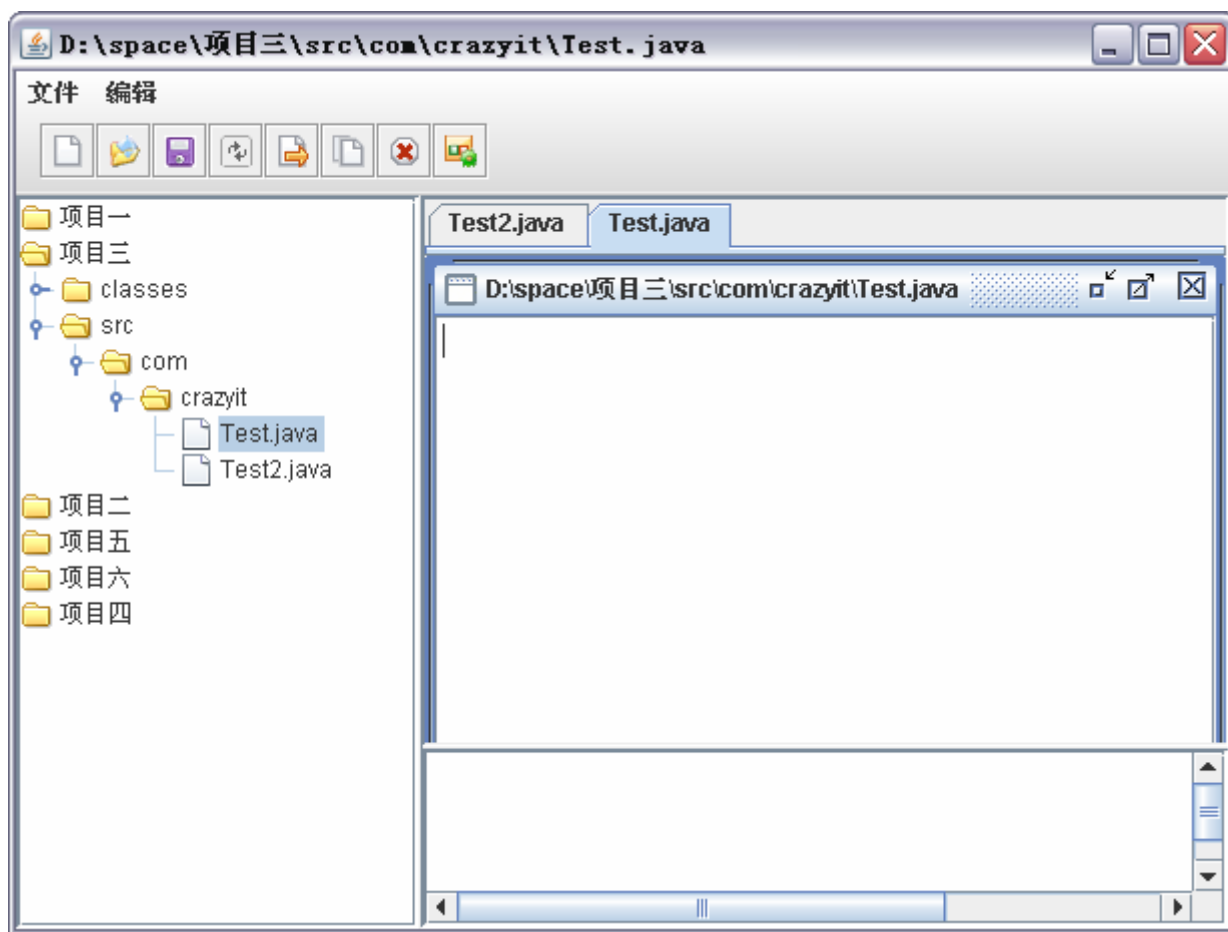


图 8.19 打开文件效果

新建了 tab 页与 EditPane 后，需要根据文件来显示对应的内容，新建一个 FileUtil 类，专门用来读取文件，并返回文件内容。

代码清单：code\editor\src\org\crazyit\editor\util\FileUtil.java

```
public class FileUtil {  
    //读取一个文件的内容  
    public static String readFile(File file) {  
        StringBuffer result = new StringBuffer();  
        FileInputStream fis= new FileInputStream(file); //使用 IO 流读取文件  
        String content = null;  
        byte[] arr = new byte[1024];  
        int readLength ;  
        while ((readLength = fis.read(arr)) > 0) {  
            content = new String(arr, 0, readLength);  
            result.append(content);  
        }  
        fis.close();  
        return result.toString();  
    }  
}
```

然后修改 EditPane 的构造器：

```
public EditPane(File file) {
```

```
//设置当前文本为文件的内容
this.setText(FileUtil.readFile(file));
}
```

重新运行就可以看到文件的内容已经被读取，并放置到对应的 **EditPane** 中。接下来，实现菜单中的打开文件功能，该功能只是通过一个文件选择器来打开文件，打开文件的操作在上面已经显示，只要通过文件选择器得到需要打开的文件，再调用 **EditorFrame** 的 **openFile** 就可以实现打开非项目中的文件了。在 **EditorFrame.java** 中新建一个 **FileChooser** 类。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```
//文件选择器
private FileChooser fileChooser;
//打开文件的 Action 对象
private Action open = new AbstractAction("打      开", new ImageIcon("images/open.gif")) {
    public void actionPerformed(ActionEvent e) {
        selectFile();
    }
};
public void selectFile() {
    fileChooser = new FileChooser(this);
}
class FileChooser extends JFileChooser {
    private EditorFrame editorFrame;
    public FileChooser(EditorFrame editorFrame){
        //调用父类的构造器
        //利用 editorFrame 的工作空间作为文件选择器打开时的默认目录
        super(editorFrame.getWorkSpace().getFolder());
        this.editorFrame = editorFrame;
        showOpenDialog(editorFrame);
    }
    public void approveSelection() {
        File file = getSelectedFile();
        //设置树当前选择的节点为 null, 让树没有被选中
        this.editorFrame.getTree().setSelectionPath(null);
        this.editorFrame.openFile(file);
        super.approveSelection();
    }
}
```

注：以上代码的黑体部分，需要设置树的当前选择节点为空，让树不选中任何节点。

实现了打开菜单的功能的后，我们需要去优化打开文件的功能，例如打开后多份文件后，需要进行文件间的切换，修改了文件后，关闭对应的 **JInternalFrame** 容器时需要弹出提示，询问是否要保存等功能。新建一个 **EditFile** 的类，表示一个编辑的文件对象。

代码清单：code\editor\src\org\crazyit\editor\commons\EditFile.java

```
public class EditFile {
    private File file; //当前编辑的文件
    private boolean saved; //该文件是否已经被保存
    private JInternalFrame iframe; //该文件对应的窗口
    private EditPane editPane; //该文件所对应的编辑器
    public EditFile(File file, boolean saved, JInternalFrame iframe,
        EditPane editPane) {
        //省略构造器代码
    }
}
```

```

    }
}

```

该类中包含了一个文件对象，一个表示是否被保存的布尔值，一个该文件对象所对应的 `JInternalFrame` 对象（该文件打开时创建的 `JInternalFrame`），还有该文件所对应的文本编辑器对象 `EditPane`。

接下来为 `EditorFrame` 添加 `EditFile` 属性，该属性表示当前正在编辑的文件对象。并为 `EditorFrame` 中新建一个集合，用于保存所有打开的文件对象。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```

private EditFile currentFile; //当前正在编辑的文件对象
private List<EditFile> openFiles = new ArrayList<EditFile>(); //打开文件的集合
//打开文件的方法
public void openFile(File file) {
    ...
    //构造一个当前修改的文件
    EditFile currentFile = new EditFile(file, true, iframe, editPane);
    //设置当前打开的文件
    this.currentFile = currentFile;
    //添加到打开文件的集合中
    this.openFiles.add(currentFile);
}
...

```

以上代码为 `EditorFrame` 设置了当前打开的文件（`currentFile`），并记录了所有打开的文件（`openFiles`）。当打开一份文件时，需要判断该份文件是否已经存在于文件列表中。如果将要被打开的文件已经被打开了，那么就切换 `tab` 页，并隐藏之前编辑的文件所对应的窗口（`JInternalFrame`），再显示将被打开的文件所对应的窗口（`JInternalFrame`）。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```

//显示 iframe 对象
public void showIFrame(JInternalFrame iframe) {
    iframe.setSelected(true);
    iframe.toFront();
}
//打开文件的方法
public void openFile(File file) {
    if (currentFile != null) {
        //如果参数 file 是当前正在编辑的文件，返回
        if (file.equals(currentFile.getFile())) return;
    }
    //在打开文件的集合中查找该文件，再判断该文件是否已经打开
    EditFile openedFile = getOpenFile(file);
    //如果文件已经打开了，调用 openExistFile 方法
    if (openedFile != null) {
        openExistFile(openedFile, file);
        return;
    }
    //打开新的文件
    openNewFile(file);
}
//从本类中记录已经打开的文件集合中得到与参数 file 一样的 EditFile 对象
private EditFile getOpenFile(File file) {

```

```

        for (EditFile openFile : openFiles) {
            if (openFile.getFile().equals(file)) return openFile;
        }
        return null;
    }
    //根据参数 file 获取该 file 对应的 tab 页中的索引
    private int getFileIndex(File file) {
        //如果从打开文件的集合中找不到与参数 file 对应的 EditFile 对象，返回-1
        EditFile openFile = getEditFile(file);
        if (openFile == null) return -1;
        return getTabIndex(openFile.getIframe().getToolTipText());
    }
    //在打开的文件中获得文件是 file 的 EditFile 对象
    private EditFile getEditFile(File file) {
        for (EditFile openFile : openFiles) {
            if (openFile.getFile().equals(file)) return openFile;
        }
        return null;
    }
    //根据字符串 tips 去 tabPane 中寻找对应的索引
    public int getTabIndex(String tips) {
        for (int i = 0; i < this.tabPane.getTabCount(); i++) {
            if (this.tabPane.getToolTipTextAt(i).equals(tips)) return i;
        }
        return -1; //参数 tips 没有在 tabPane 中，返回-1
    }
    //打开已经存在的文件（已经在编辑文件集中的文件），openFiles 集合中的文件
    public void openExistFile(EditFile openedFile, File willOpenFile) {
        //将 tab 页变成当前所选择文件的索引
        tabPane.setSelectedIndex(getFileIndex(willOpenFile));
        //显示 iframe
        showIframe(openedFile.getIframe());
        //设置当前打开文件
        this.currentFile = openedFile;
        //添加到当前打开的文件集合中
        this.openFiles.add(openedFile);
    }
    //打开一个不在编辑集中的文件
    public void openNewFile(File file) {
        setTitle(file.getAbsolutePath()); //设置 EditorFrame 的标题为该文件的全路径
        //创建一个 JInternalFrame 对象，title 为文件的绝对路径
        JInternalFrame iframe = new JInternalFrame(file.getAbsolutePath(), true, true, true, true);
        EditPane editPane = new EditPane(file); //新建一个 EditPane 对象
        iframe.add(new JScrollPane(editPane));
        desk.add(iframe);
        iframe.show();
        iframe.reshape(0, 0, 400, 300);
        tabPane.addTab(file.getName(), null, null, file.getAbsolutePath());
        tabPane.setSelectedIndex(tabPane.getTabCount() - 1);
        //设置当前的文件对象

```



```

        this.currentFile = new EditFile(file, true, iframe, editPane);
        //将当前的文件加到打开的文件集合中
        this.openFiles.add(this.currentFile);
    }

```

注意，以上代码的黑体部分，判断将被打开的文件是否已经被打开（存在于 `openFiles` 集合中），如果已经打开了，则调用 `openExistFile` 方法，如果没有被打开，则调用 `openNewFile` 方法。`openExistFile` 方法主要是从 `openFiles` 集合中获取对应的 `EditFile` 对象，得到该对象后可以得到对应的 `JInternalFrame` 并控制窗口显示即可，最后设置当前打开的文件（`currentFile`）。`openNewFile` 方法主要负责创建窗口，设置当前打开文件的对象，再将该对象放到 `openFiles` 集合中。

实现了文件的打开后，运行 `Main` 类可看到效果，可以发现点击 `tab` 页时，对应的窗口并没有发生改变，我们可以为 `tab` 页（`tabPane`）添加监听器 `TabListener`，当点击了 `tab` 时，使其发生相应的改变。

代码清单：code\editor\src\org\crazyit\editor\TabListener.java

```

public class TabListener implements ChangeListener {
    //主界面的 IFrame
    private EditorFrame editorFrame;
    public TabListener(EditorFrame editorFrame) {
        this.editorFrame = editorFrame;
    }
    public void stateChanged(ChangeEvent e) {
        //获得当前点击 tab 页对象
        JTabbedPane tab = (JTabbedPane)e.getSource();
        //获得 tab 页的索引
        int index = tab.getSelectedIndex();
        if (index == -1) return;
        //根据 tab 页的 tips(文件的绝对路径)获得当前的 JInternalFrame 对象
        JInternalFrame currentFrame = editorFrame.getIFrame(tab.getToolTipTextAt(index));
        //让当前点击的 JInternalFrame 对象可见
        editorFrame.showIFrame(currentFrame);
        //根据当前的 JInternalFrame 对象获得对应的文件
        EditFile currentFile = editorFrame.getEditFile(currentFrame);
        //设置 EditorFrame 当前编辑的文件为 tab 对应的文件
        editorFrame.setCurrentFile(currentFile);
    }
}

```

以上代码的黑体部分，在 `EditorFrame` 中实现这两个方法。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```

//根据 JInternalFrame 标题找到 JInternalFrame 对象
public JInternalFrame getIFrame(String title) {
    JInternalFrame[] iframes = desk.getAllFrames();
    for (JInternalFrame iframe : iframes) {
        if (iframe.getTitle().equals(title)) return iframe;
    }
    return null;
}
//根据 JInternalFrame 在打开的文件集合中获取相应的文件对象
public EditFile getEditFile(JInternalFrame iframe) {
    for (EditFile openFile : openFiles) {
        if (openFile.getIframe().equals(iframe)) return openFile;
    }
}

```

```

        return null;
    }
}

```

实现了 **tab** 页的监听器后，为 **tabPane** 对象设置监听器。在 **EditorFrame** 的 **addListeners** 方法中加入如下代码：

```
tabPane.addChangeListener(new TabListener(this)); //为 tabPane 添加监听器
```

运行 **Main** 类可看到效果，点击 **tab** 页，可看到窗口间的切换。接下来实现窗口对象的监听器，当窗口被激活或者被关闭时，需要添加一些行为，例如切换 **tab** 页面，关闭时弹出询问是否要保存等行为。新建一个 **IFrameListener** 类，作为窗口的监听器。

代码清单：code\editor\src\org\crazyit\editor\IFrameListener.java

```

public class IFrameListener extends InternalFrameAdapter {
    //主界面对象
    private EditorFrame editorFrame;
    public IFrameListener(EditorFrame editorFrame) {
        this.editorFrame = editorFrame;
    }
    //窗口激活执行的方法
    public void internalFrameActivated(InternalFrameEvent e) {
        JInternalFrame iframe = editorFrame.getDesk().getSelectedFrame();
        int tapIndex = editorFrame.getTabIndex(iframe.getTitle());
        editorFrame.getTabPane().setSelectedIndex(tapIndex);
    }
    //窗口关闭执行的方法
    public void internalFrameClosing(InternalFrameEvent e) {
        //获取当前关闭操作所对应的 JInternalFrame
        JInternalFrame iframe = (JInternalFrame)e.getSource();
        //获取当前操作的文件
        EditFile editFile = editorFrame.getCurrentFile();
        //询问是否要保存
        editorFrame.askSave(editFile);
        //关闭当前的 iframe
        editorFrame.closeIFrame(iframe);
    }
}

```

注意上面代码中的黑体部分，先在窗口关闭的时候询问是否要保存该文件，再执行关闭窗口的方法，我们在 **EditorFrame** 中实现这两个方法。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```

//询问是否要保存，参数为将被打开的文件
public void askSave(EditFile file) {
    //该份文件修改过没有保存
    if (!file.isSaved()) {
        //弹出询问
        int val = JOptionPane.showConfirmDialog(this, "是否要保存？", "询问",
            JOptionPane.YES_NO_OPTION);
        //点击了需要保存
        if (JOptionPane.YES_OPTION == val) {
            //调用 EditorFrame 的保存方法对文件进行保存
            saveFile(file);
        }
    }
}
}

```

```

//保存文件的方法
public void saveFile(EditFile file) {
    //暂时提供空实现
}
//关闭一个窗口
public void closeFrame(JInternalFrame iframe) {
    //获得当前的文件，即要关闭的文件对象
    EditFile closeFile = getEditFile(iframe);
    //设置本类中的 currentFile 属性
    afterClose(closeFile);
    //获得该 iframe 在 tab 页中对应的索引
    int index = getTabIndex(iframe.getTitle());
    //从 tab 页中删除
    getTabPane().remove(index);
    //从打开的文件集合中删除这个关闭的文件
    openFiles.remove(closeFile);
}
//当关闭一份文件后，设置本对象的 currentFile 属性
private void afterClose(EditFile closeFile) {
    //获取关闭文件在打开文件集合中的索引
    int openFilesIndex = getEditFileIndex(closeFile);
    //如果该文件已经是所有打开的文件最后一份
    if (this.openFiles.size() == 1) {
        this.currentFile = null;
    } else { //如果还有其他文件，判断关闭的文件位置
        if (openFilesIndex == 0) {
            //如果关闭的文件是第一份，拿集合中的第二份
            this.currentFile = openFiles.get(openFilesIndex + 1);
        } else if (openFilesIndex == (openFiles.size() - 1)) {
            //如果关闭的是最后一份，取倒数第二份
            this.currentFile = openFiles.get(openFiles.size() - 2);
        } else {
            //不是第一份，也不是最后一份
            this.currentFile = openFiles.get(openFilesIndex - 1);
        }
    }
}
//获取 editFile 在打开的文件集合中的索引
private int getEditFileIndex(EditFile editFile) {
    for (int i = 0; i < this.openFiles.size(); i++) {
        if (openFiles.get(i).equals(editFile)) return i;
    }
    return -1;
}

```

注：代码中的 askSave 方法，是否弹出询问是根据当前编辑文件（EditFile）的 saved 属性决定，如果需要保存，则执行 saveFile 方法（以上代码的黑体部分），该方法暂时提供空实现。我们将在后面的章节实现它。

在 EditorFrame 的构造器中创建 IFrameListener 对象，并在 openNewFile 方法中为新建的 JInternalFrame 对象添加监听器。

```
//为 JInternalFrame 添加窗口监听器
```

```
iframe.addInternalFrameListener(this.iframeListener);
```

实现了窗口的监听器后，可以运行 **Main** 类，点击项目树上的各个节点，可以看到 **tab** 页与显示的内容随着点击而做相应的切换，这就是我们所需要的效果，但是，即使对文件进行了修改，在关闭窗口的时候，仍然无法弹出询问，这是由于 **EditFile** 的 **saved** 从没有被改变过，在创建这个对象的时候，该属性就一直是 **true**，我们需要为 **EditPane** 添加一个监听器，在文本的内容发生改变时，设置 **EditFile** 的 **saved** 属性为 **false**。新建一个 **EditDocumentListener** 类。

代码清单：code\editor\src\org\crazyit\editor\EditDocumentListener.java

```
public class EditDocumentListener implements DocumentListener {
    //主界面对象
    private EditorFrame editorFrame;
    public EditDocumentListener(EditorFrame editorFrame) {
        this.editorFrame = editorFrame;
    }
    public void changedUpdate(DocumentEvent e) {}
    public void insertUpdate(DocumentEvent e) {
        this.editorFrame.getCurrentFile().setSaved(false); //设置当前编辑的文件的 saved 属性为 false
    }
    public void removeUpdate(DocumentEvent e) {}
}
```

代码中的黑体部分，设置当前编辑文件的 **saved** 属性为 **false**。同时修改 **EditFrame** 的 **openNewFile** 方法。代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```
//打开一个不在编辑集合中的文件
public void openNewFile(File file) {
    ...
    EditPane editPane = new EditPane(file); //新建一个 EditPane 对象
    //为 EditPane 添加键盘监听器
    editPane.getDocument().addDocumentListener(new EditDocumentListener(this));
    ...
}
```

重新运行 **Main** 类，对打开的一份文件作出修改，可以看到弹出的提示。由于我们保存文件的方法暂时是空实现，因此确定保存没有任何效果。

### 8.6.6 显示行数与高亮

在 8.6.5 中我们实现了文件的打开功能，并可以在 **EditPane** 中显示对应的内容，对于一些常见的 IDE，如 **Eclipse**、**EditPlus** 等，都有有显示行数，高亮显示的功能，本小节实现这两个常见的功能。先去实现比较简单的行号功能。

代码清单：code\editor\src\org\crazyit\editor\EditPane.java

```
protected StyledDocument doc; //样式文档对象
private SimpleAttributeSet lineAttr = new SimpleAttributeSet(); //定义文档中行数文本的外观属性
public EditPane(File file) {
    this.setText(FileUtil.readFile(file));
    this.doc = getStyledDocument(); //获取这个编辑器的模型文档
    this.setMargin(new Insets(3, 40, 0, 0)); //设置该文档的页边距
}
//重画该组件，设置行号
public void paint(Graphics g){
    super.paint(g);
```

```

Element root = doc.getDefaultRootElement();
int line = root.getElementIndex(doc.getLength()); //获得行号
g.setColor(new Color(230, 230, 230)); //设置颜色
g.fillRect(0, 0, this.getMargin().left - 10, getSize().height); //绘制行数矩形框
g.setColor(new Color(40, 40, 40)); //设置行号的颜色
//每行绘制一个行号
for (int count = 0, j = 1; count <= line; count++, j++) {
    g.drawString(String.valueOf(j), 3,
        (int)((count + 1) * 1.5020 * StyleConstants.getFontSize(lineAttr)));
}
}

```

运行 Main 并打开一份文件，可以看到效果如图 8.20 所示。

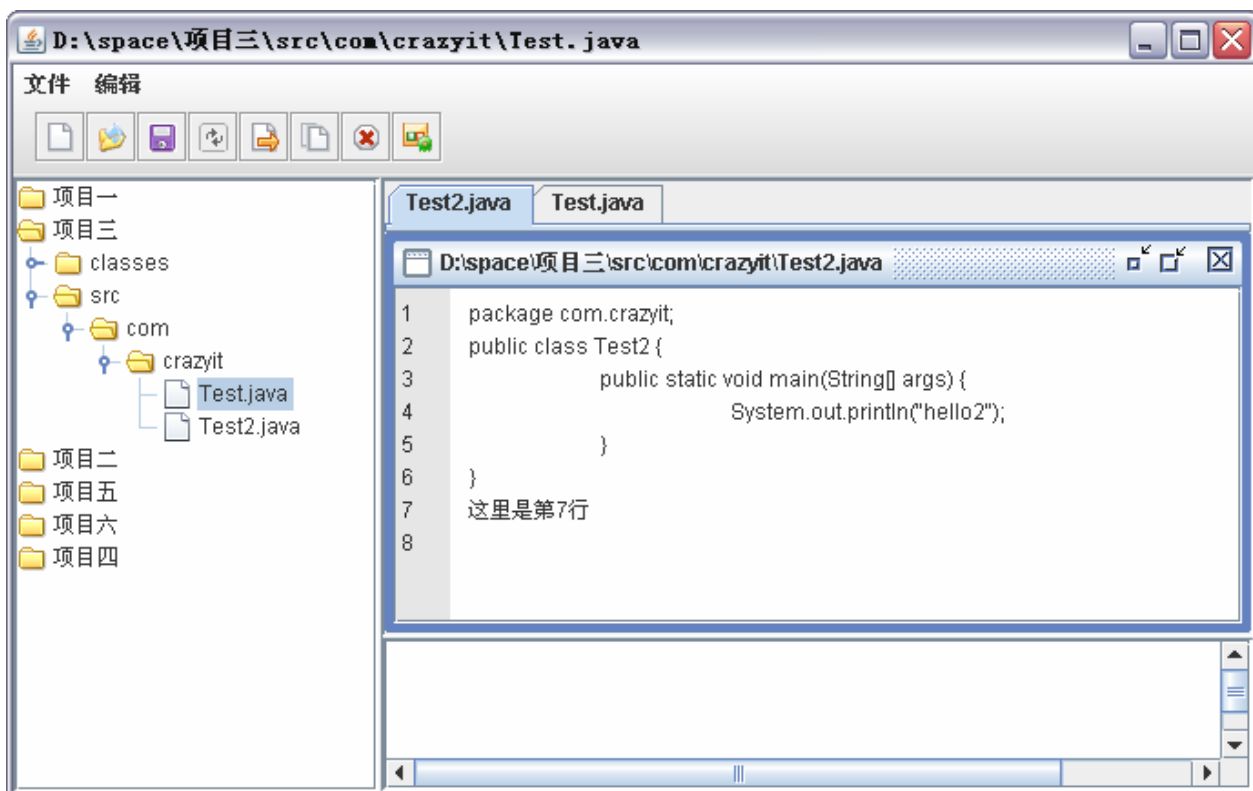


图 8.20 实现显示行数功能

下面去实现高亮功能，高亮功能，也就是将某些常用的关键字使用不同的颜色显示在文本编辑器中。在 `EditPane.java` 中新建一个 `SyntaxFormatter` 类，用于读取 `stx` 文件，使用 `Scanner` 类进该文件进行分析，将 `stx` 文件中定义的颜色与关键字关联起来，在 `EditPane` 中对它里面的文本进行分析，遇到某些关键字，再使用 `SyntaxFormatter` 进行高亮显示。

代码清单：code\editor\src\org\crazyit\editor\EditPane.java

```

class SyntaxFormatter {
    //以一个 Map 保存关键字和颜色的对应关系
    private Map<SimpleAttributeSet, ArrayList> attMap
        = new HashMap<SimpleAttributeSet, ArrayList>();
    //定义文档的正常文本的外观属性
    SimpleAttributeSet normalAttr = new SimpleAttributeSet();
    public SyntaxFormatter(String syntaxFile) {
        //设置正常文本的颜色、大小
    }
}

```

```

StyleConstants.setForeground(normalAttr, Color.BLACK);
//StyleConstants.setFontSize(normalAttr, 14);
//创建一个 Scanner 对象，负责根据语法文件加载颜色信息
Scanner scanner = null;
scanner = new Scanner(new File(syntaxFile));
int color = -1;
ArrayList<String> keywords = new ArrayList<String>();
//不断读取语法文件的内容行
while(scanner.hasNextLine()) {
    String line = scanner.nextLine();
    //如果当前行以#开头
    if (line.startsWith("#")) {
        if (keywords.size() > 0 && color > -1) {
            //取出当前行的颜色值，并封装成 SimpleAttributeSet 对象
            SimpleAttributeSet att = new SimpleAttributeSet();
            StyleConstants.setForeground(att, new Color(color));
            //StyleConstants.setFontSize(att, 14);
            //将当前颜色和关键字 List 对应起来
            attMap.put(att, keywords);
        }
        //重新创建新的关键字 List，为下一个语法格式准备
        keywords = new ArrayList<String>();
        color = Integer.parseInt(line.substring(1), 16);
    } else {
        //对于普通行，每行内容添加到关键字 List 里
        if (line.trim().length() > 0) keywords.add(line.trim());
    }
}
//把最后的关键字和颜色对应起来
if (keywords.size() > 0 && color > -1) {
    SimpleAttributeSet att = new SimpleAttributeSet();
    StyleConstants.setForeground(att, new Color(color));
    attMap.put(att, keywords);
}
}
//返回该格式器里正常文本的外观属性
public SimpleAttributeSet getNormalAttributeSet() {
    return normalAttr;
}
//设置语法高亮
public void setHighLight(StyledDocument doc, String token,
    int start, int length) {
    //保存需要对当前单词对应的外观属性
    SimpleAttributeSet currentAttributeSet = null;
    outer :
    for (SimpleAttributeSet att : attMap.keySet()) {
        //取出当前颜色对应的所有关键字
        ArrayList keywords = attMap.get(att);
        //遍历所有关键字
        for (Object keyword : keywords) {

```



}

果需要进行高亮显示，则调用 `setHighLight`（以上代码的黑体字）方法即可。下面为 `EditPane` 类添加一个分析文本的方法。

代码清单: code\editor\src\org\crazyit\editor\EditPane.java

```
protected StyledDocument doc; //样式文档对象
protected SyntaxFormatter formatter = new SyntaxFormatter("java.stx");//构造一个文本格式器
private SimpleAttributeSet quotAttr = new SimpleAttributeSet();
private int docChangeStart = 0; //保存文档改变的开始位置
private int docChangeLength = 0; //保存文档改变的长度
private SimpleAttributeSet lineAttr = new SimpleAttributeSet();//定义文档中行数文本的外观属性
//分析关键字方法
public void syntaxParse() {
    Element root = doc.getDefaultRootElement();//获取文档的根元素，即文档内的全部内容
    int cursorPos = this.getCaretPosition();//获取文档中光标插入符的位置
    int line = root.getElementIndex(cursorPos);
    Element para = root.getElement(line); //获取光标所在位置的行
    int start = para.getStartOffset();//定义光标所在行的行头在文档中位置
    //如果文档修改位置比当前行还前
    if (start > docChangeStart) start = docChangeStart;
    //定义被修改部分的长度
    int length = para.getEndOffset() - start;
    if (length < docChangeLength) length = docChangeLength + 1;
    //取出所有被修改的字符串
    String s = doc.getText(start, length);
    //以空格、点号等作为分隔符
    String[] tokens = s.split("\\s+|\\.|\\(|\\)|\\{\\}|\\[\\]|\"");
    //定义当前分析单词的在 s 字符串中的开始位置
    int curStart = 0;
    boolean isQuot = false;
    for (String token : tokens) {
        //找出当前分析单词在 s 字符串的中位置
        int tokenPos = s.indexOf(token , curStart);
        if (isQuot && (token.endsWith("\"") || token.endsWith("\'"))) {
```

```

        doc.setCharacterAttributes(start + tokenPos, token.length(), quotAttr, false);
        isQuot = false;
    } else if (isQuot && !(token.endsWith("\"") || token.endsWith("\'"))) {
        doc.setCharacterAttributes(start + tokenPos, token.length(), quotAttr, false);
    } else if ((token.startsWith("\"") || token.startsWith("\'"))
        && (token.endsWith("\"") || token.endsWith("\'"))) {
        doc.setCharacterAttributes(start + tokenPos, token.length(), quotAttr, false);
    } else if ((token.startsWith("\"") || token.startsWith("\'"))
        && !(token.endsWith("\"") || token.endsWith("\'"))) {
        doc.setCharacterAttributes(start + tokenPos, token.length(), quotAttr, false);
        isQuot = true;
    } else {
        //使用格式器对当前单词设置颜色
        formatter.setHighLight(doc, token, start + tokenPos, token.length());
    }
    //开始分析下一个单词
    curStart = tokenPos + token.length();
}
}

```

以上代码中，`syntaxParse` 方法对文本进行分析，如果是关键字，则使用 `SyntaxFormatter` 格式器对文本进行高亮显示。修改 `EditPane` 的构造器，当打开文件的时候，就对文件内容进行高亮显示。

代码清单：code\editor\src\org\crazyit\editor\EditPane.java

```

public EditPane(File file) {
    this.setText(FileUtil.readFile(file));
    this.doc = getStyledDocument();
    this.setMargin(new Insets(3, 40, 0, 0)); //设置该文档的页边距
    syntaxParse();
    //添加按键监听器，当按键松开时进行语法分析
    this.addKeyListener(new KeyAdapter() {
        public void keyReleased(KeyEvent ke) {
            syntaxParse();
        }
    });
}

```

在文件打开的时候（构造 `EditPane` 的时候），就对文本进行分析并进行高亮显示，在编辑的时候，进行键盘点击就进行语法分析。运行 `Main` 类，可以看到效果如图 8.21 所示。

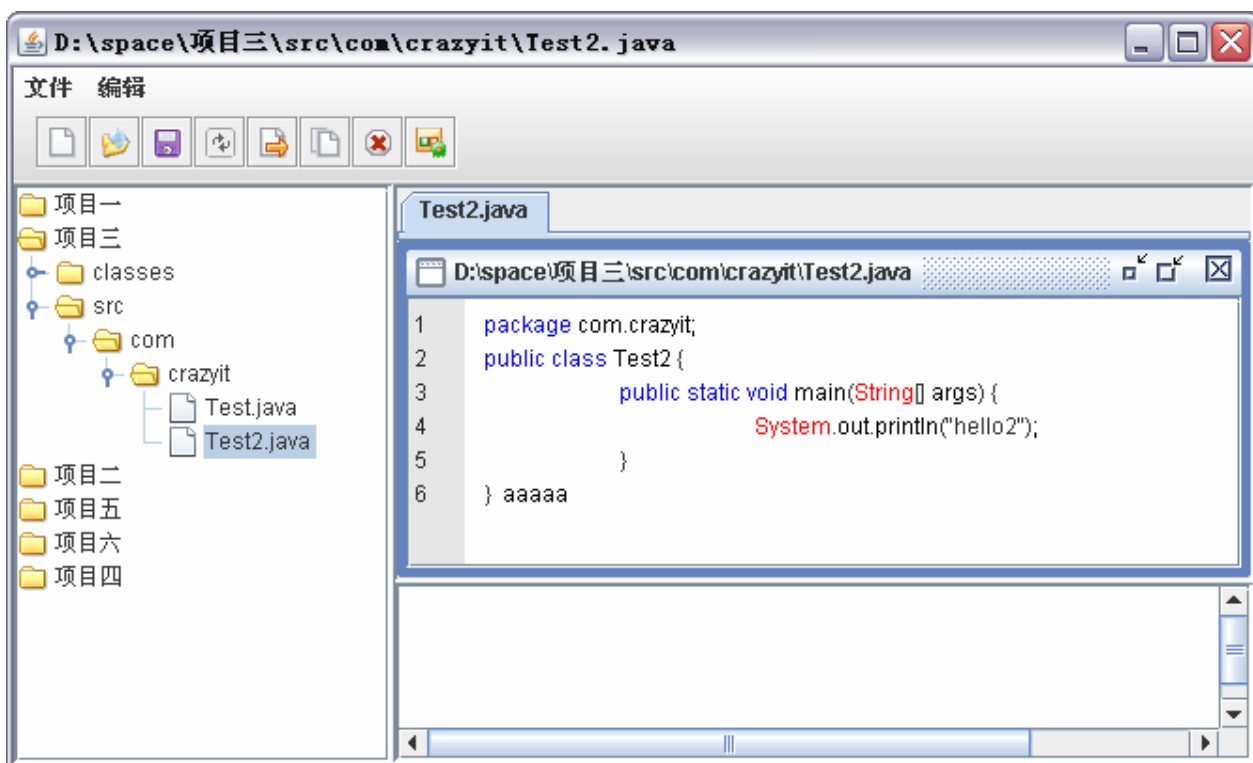


图 8.21 实现高亮显示功能

到这里，显示行号和关键字高亮显示的功能已经完成，接下来实现文件的操作。

### 8.6.7 文件的保存功能

在 `EditorFrame` 中有一个 `saveFile` 的方法，在 8.6.5 中，当对文件进行了修改，就弹出询问，是否需要保存文件，当时尚未实现 `saveFile` 方法，现在对 `saveFile` 作一个简单的实现。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```
//保存文件的方法
public void saveFile(EditFile file) {
    //使用 FileUtil 的方法将 EditFile 对应的编辑器的内容写进文件
    FileUtil.writeFile(file.getFile(), file.getEditPane().getText());
    //设置操作的 EditFile 对象已经被保存
    file.setSaved(true);
}
```

为 `FileUtil` 类新增一个 `writeFile` 方法，使用 IO 流将内容写到文件中。

代码清单：code\editor\src\org\crazyit\editor\util\FileUtil.java

```
public static void writeFile(File file, String content) {
    FileOutputStream fos = new FileOutputStream(file);
    PrintStream ps = new PrintStream(fos);
    ps.print(content);
    ps.close();
    fos.close();
}
```

这样就可以实现文件的保存，但是，在实际情况下，如果要保存一份 Java 文件的话，保存的时候还需要将该份文件编译到相应的目录中。我们新建一个 `SaveHandler` 接口，里面有一个 `save` 的方法，

返回值是保存文件时的信息，新建一个 `CommonSaveHandler` 的类继承 `SaveHandler` 作为处理普通保存，再新建一个 `JavaSaveHandler` 继承 `CommonSaveHandler` 类，该类处理保存 Java 文件。

`SaveHandler` 接口：

```
String save(EditorFrame editorFrame); //设置该文档的页边距
```

`CommonSaveHandler` 实现 `SaveHandler` 接口。

代码清单：code\editor\src\org\crazyit\editor\handler\save\CommonSaveHandler.java

```
//提供一个保存方法，作为普通的保存
public String save(EditorFrame editorFrame) {
    EditFile editFile = editorFrame.getCurrentFile();
    FileUtil.writeFile(editFile.getFile(), editFile.getEditPane().getText());
    return null;
}
```

再为 `EditorFrame` 提供一个方法，用于返回当前选择节点所属的项目的目录对象。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```
//返回项目树当前所选中的节点所属的项目节点对应的目录
public File getCurrentProject() {
    //获取根节点(工作空间)
    ProjectTreeNode root = (ProjectTreeNode) getSelectNode().getRoot();
    //获取根节点下的所有子节点（即项目节点集合）
    List<ProjectTreeNode> projects = root.getChildren();
    ProjectTreeNode selectNode = getSelectNode();
    if (selectNode != null) {
        for (ProjectTreeNode project : projects) {
            //当前先中节点是该 project 下的子节点
            if (selectNode.isNodeAncestor(project)) return project.getFile();
        }
    }
    return null;
}
```

上面的代码得到当前节点所属的项目目录，可以通过这个项目的目录获取该项目的编译路径，保存 Java 文件的时候，就可以将源文件编译到指定的目录中。编译 Java 文件，我们需要 `Runtime` 类的 `exec` 方法执行一个 Java 的编译命令，该方法返回一个 `Process` 对象，就是执行这个命令时的进程，我们可以通过这个进程的错误信息，看下我们的 Java 文件编译是否正确。编写一个 `CommandUtil` 的类，用于返回一个进程（`Process`）的错误信息。

代码清单：code\editor\src\org\crazyit\editor\util\CommandUtil.java

```
//返回进程 process 的错误信息
public static String getErrorMessage(Process process) {
    return getProcessString(process.getErrorStream());
}

private static String getProcessString(InputStream is) {
    StringBuffer buffer = new StringBuffer();
    byte[] b = new byte[4096];
    for (int n; (n = is.read(b)) != -1;) buffer.append(new String(b, 0, n));
    is.close();
    return buffer.toString();
}

//获取一个进程的错误信息
public static String getProcessString(Process process) {
    StringBuffer result = new StringBuffer();
```

```

        String errorString = getErrorMessage(process); //调用 CommandUtil 的方法
        if (errorString.length() != 0) result.append("错误: " + errorString);
        else result.append("执行完成");
        return result.toString();
    }
    //返回一个进程的信息
    public static String getRunString(Process process) {
        String error = getErrorMessage(process);
        String message = getProcessString(process.getInputStream());
        return error + message;
    }
}

```

下面可以去实现 `JavaSaveHandler` 类。

代码清单: `code\editor\src\org\crazyit\editor\handler\save\JavaSaveHandler.java`

```

public String save(EditorFrame editorFrame) {
    //调用父类的保存方法
    super.save(editorFrame);
    return javac(editorFrame);
}
//执行 javac 命令
private String javac(EditorFrame editorFrame) {
    //获得项目的编译路径, 项目目录加 CompileConfig 中配置的输出目录
    String classPath = editorFrame.getCurrentProject().getAbsolutePath()
        + File.separator + CompileConfig.OUTPUT_DIR;
    //获得源文件的文件路径
    String filePath = editorFrame.getCurrentFile().getFile().getAbsolutePath();
    //拼装字符串命令, 该命令只可在 windows 下运行
    String command = "cmd /c javac -d " + classPath + " " + filePath;
    Process process = Runtime.getRuntime().exec(command); //执行命令
    process.waitFor(); //等待进程运行结束
    return CommandUtil.getProcessString(process); //返回进程的信息
}

```

这里需要注意的是, 在 **Windows** 系统中, 如果文件目录中有空格, 那么就不会正常编译, 因此我们可以将这些拼装的命令放到一个 **bat** 文件中, 再对有空格的目录加入冒号, 这样就可以避免这些问题。为 `CommandUtil` 加入一个工具方法。

代码清单: `code\editor\src\org\crazyit\editor\util\CommandUtil.java`

```

public static Process executeCommand(String command) {
    //在 windows 下将命令生成一份 bat 文件, 再执行该 bat 文件
    File batFile = new File("dump.bat");
    if (!batFile.exists()) batFile.createNewFile();
    //将命令写入文件中
    FileWriter writer = new FileWriter(batFile);
    writer.write(command);
    writer.close();
    //执行该 bat 文件
    Process process = Runtime.getRuntime().exec(WINDOWS_COMMAND +
        batFile.getAbsolutePath());
    process.waitFor();
    //将 bat 文件删除
    batFile.delete();
    return process;
}

```

```
}
```

那么可以修改编译 Java 文件方法，直接调用工具类中的 `executeCommand` 方法：

```
//拼装字符串命令，该命令只可在 windows 下运行
String command = "javac -d \"" + classPath + "\" \"" + filePath + "\"";
Process p = CommandUtil.executeCommand(command);
return CommandUtil.getProcessString(p);
```

`JavaSaveHandler` 继承于 `CommonSaveHandler` 类，并在该类的基础上运行 `javac` 命令，编译 java 文件。如果直接在 `EditorFrame` 中使用 `SaveHandler` 对象，那么将会多加许多的判断，判断文件是否为 Java 文件，再用不同的 `SaveHandler` 类去处理保存动力，我们可以使用设计模式中的中介者模式，来减低这样的代码耦合。新建一个中介者抽象类 `SaveMediator`。并为其新增一个子类，实现 `SaveMediator` 的 `doSave` 方法。

代码清单：code\editor\src\org\crazyit\editor\handler\save\SaveMediator.java

```
//需要子类去实现的保存方法
public abstract String doSave(EditorFrame editorFrame);
```

代码清单：code\editor\src\org\crazyit\editor\handler\save\SaveMediatorConcrete.java

```
private SaveHandler commonHandler;
private SaveHandler javaHandler;
//构造两个处理保存的对象
public SaveMediatorConcrete() {
    this.commonHandler = new CommonSaveHandler();
    this.javaHandler = new JavaSaveHandler();
}
public String doSave(EditorFrame editorFrame) {
    //获得当前编辑的文件名
    String fileName = editorFrame.getCurrentFile().getFile().getName();
    String result = null;
    //判断文件是否为 Java 文件， 再决定处理类
    if (fileName.endsWith(".java")) { //保存 java 文件
        result = javaHandler.save(editorFrame);
    } else { //执行普通的保存
        result = commonHandler.save(editorFrame);
    }
    return result;
}
```

这样，我们就定义了一个中介者的角色，再去修改 `EditorFrame` 中保存文件的方法即可。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```
private SaveMediator saveMediator; //中介者对象
//用于保存当前所打开的文件
public void saveFile(EditFile file) {
    if (file == null) return;
    //调用中介者对象的方法去保存文件
    String result = saveMediator.doSave(this);
    //将结果放到信息显示区的文本域中
    infoArea.setText(result);
    //写完文件后，设置当前文件的保存状态为 true，表示已经保存
    file.setSaved(true);
}
```

上面的中介者对象，需要在构造器中初始化，并在相应的监听器中调用 `saveFile` 方法。运行 `Main` 类，并新建一份 Java 文件进行保存，效果如图 8.22。



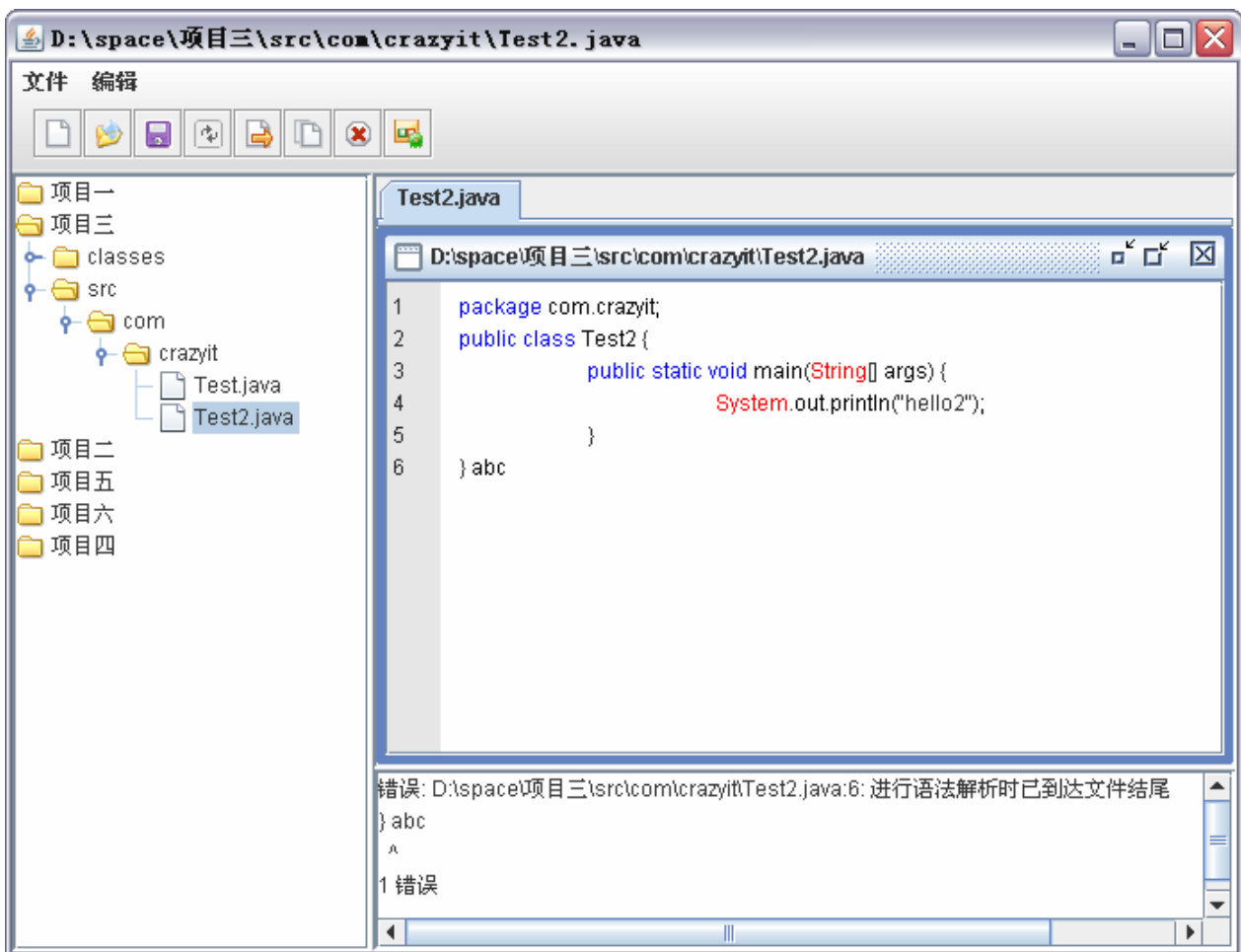


图 8.22 进行 Java 文件保存

文件的保存操作已经完成，在实现的过程中，我们使用了设计模式中的中介者模式，降低了 `EditorFrame` 与保存处理类之间的代码耦合。

### 8.6.7 运行Java文件

在 8.6.6 中我们实现了保存并编译 Java 文件的功能，这一节将实现运行 Java 文件的功能，运行与编译的原理一样，都是使用 `Runtime` 中的 `exec` 方法来执行 `java` 命令即可。新增一个 `JavaRunHandler` 的类，用于运行 Java 编译后的 class 文件。

代码清单：code\editor\src\org\crazyit\editor\handler\run\JavaRunHandler.java

```
public String run(EditorFrame editorFrame) {
    //获得项目目录的路径
    String projectPath = editorFrame.getCurrentProject().getAbsolutePath();
    //获得源文件的全路径
    String sourcePath = editorFrame.getCurrentFile().getFile().getAbsolutePath();
    //获得项目的编译路径，项目目录加 CompileConfig 中配置的输出目录
    String classPath = editorFrame.getCurrentProject().getAbsolutePath()
        + File.separator + CompileConfig.OUTPUT_DIR;
    String className = getClassName(projectPath, sourcePath); //获取类名
    //拼装命令
```

```
String command = "java -cp \"" + classPath + "\" " + className;
Process p = CommandUtil.executeCommand(command);
return CommandUtil.getRunString(p);
}
//根据项目目录的路径和 Java 源文件的路径获取一个类的全限定类名
private String getClassName(String projectPath, String sourcePath) {
    String temp = projectPath + File.separator + CompileConfig.SRC_DIR +
    File.separator;
    String result = sourcePath.replace(temp, "");
    result = result.replace(".java", "");
    result = result.replace(String.valueOf(File.separatorChar), ".");
    return result;
}
```

Java 运行类文件的命令是：java -cp classpath package + className。为 EditorFrame 类新增一个 run 方法，用于运行文件（当前只能运行 class 文件），再去实现运行的 Action。

代码清单：code\editor\src\org\crazyit\editor\EditorFrame.java

```
//运行 class 文件的处理类
private JavaRunHandler runHandler;
//运行文件的方法
public void run() {
    saveFile(getCurrentFile()); //运行前先保存
    String result = runHandler.run(this); //将结果显示
    infoArea.setText(result);
}
```

注：runHandler 需要在 EditorFrame 的构造器中初始化。

在相应的监听器中调用 run 方法即可。运行 Main 类，运行 Java 文件，得到的效果如图 8.23。

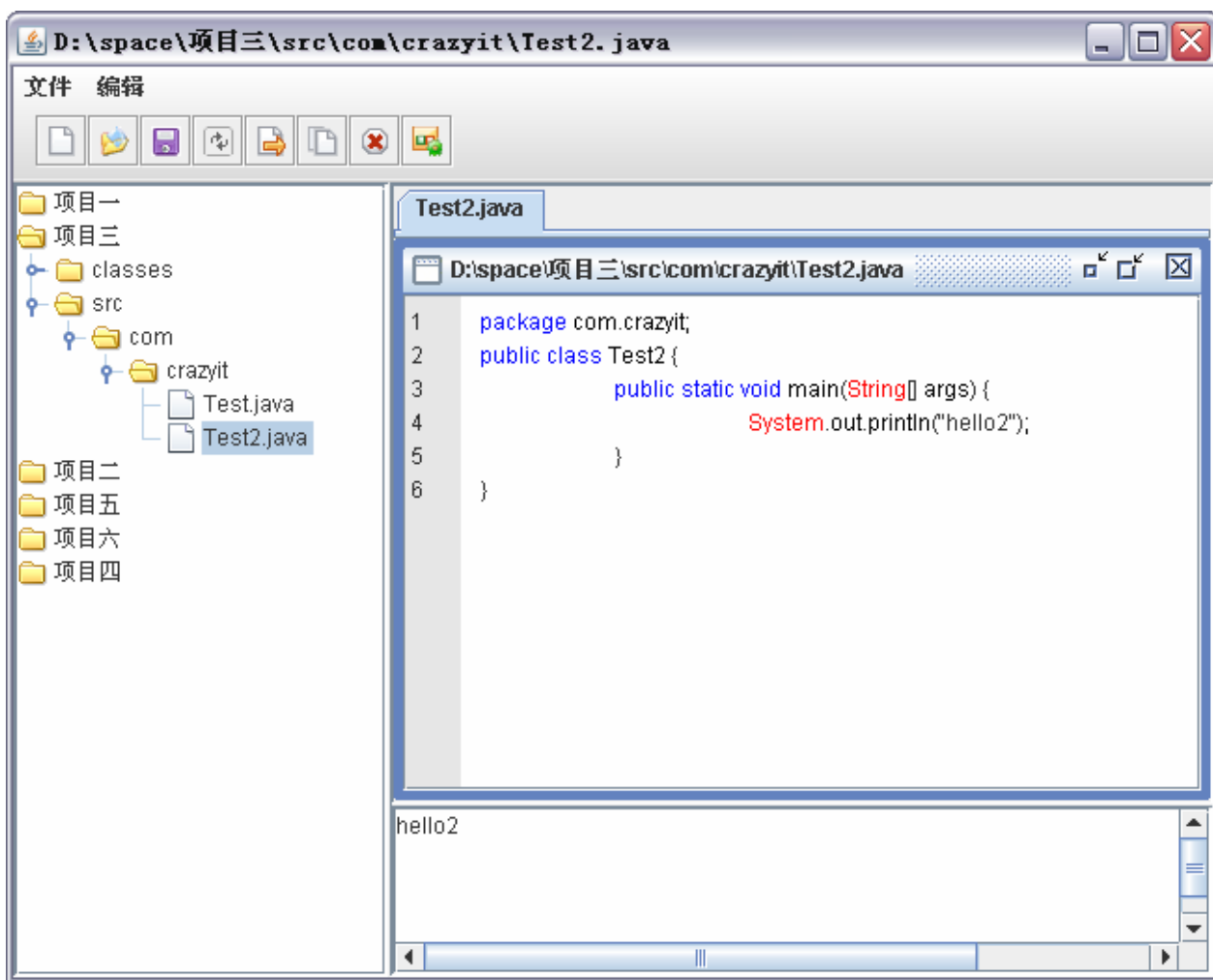


图 8.23 实现运行 Java 文件功能

8.6 已经全部实现了我们在本章开头所定义的文件操作的功能，包括文件的新建，打开，保存，运行，项目树的刷新等。

## 8.7 实现文本操作

本例中文本操作主要包括复制、粘贴和剪切这三个功能，在建立界面的时候，就已经建立了相对应的 **Action** 对象，只是尚未实现，本小节对它们进行实现。

### 8.7.1 文本操作

实现复制、剪切和粘贴功能都非常简单，**JTextPane** 已经帮我们实现好了，我们只需要拿回当前正在编辑的 **EditPane**（继承于 **JTextPane**）对象即可做到。

实现复制的 **Action**：

//复制文本的 **Action** 对象

```
private Action copy = new AbstractAction("复制", new ImageIcon("images/copy.gif")) {
    public void actionPerformed(ActionEvent e) {
```

```
        if (getCurrentFile() != null) {  
            getCurrentFile().getEditPane().copy();  
        }  
    }  
};
```

实现剪切的 Action:

```
//剪切文本的 Action 对象  
private Action cut = new AbstractAction("剪    切", new ImageIcon("images/cut.gif")) {  
    public void actionPerformed(ActionEvent e) {  
        if (getCurrentFile() != null) getCurrentFile().getEditPane().cut();  
    }  
};
```

实现粘贴的 Action:

```
//粘贴文本的 Action 对象  
private Action paste = new AbstractAction("粘    贴", new ImageIcon("images/paste.gif")) {  
    public void actionPerformed(ActionEvent e) {  
        if (getCurrentFile() != null) getCurrentFile().getEditPane().paste();  
    }  
};
```

到此，文本操作的相对应的 Action 都已经实现。

## 8.8 本章小节

本章实现了一个简单的 Java IDE 工具，基本的功能有文件的操作，文本的操作，显示行数的功能，高亮显示关键字的功能，分析语法的功能，在实现这些功能的过程中，详细介绍了上述功能的实现方法与实现原理，并使用了策略模式、中介者模式等设计模式，主要讲述了 JTree 的使用，如何利用 JTree 实现创建项目树，刷新项目树等功能。

## 第9章 图书进存销系统

### 9.1 项目简介

本章介绍如何开发一个 CS 结构的图书进存销系统，该系统的主要有出版社管理、书本管理、书的入库管理、销售管理等功能，通过这些简单的功能，可以让我们了解如何利用 JDBC 进行数据库操作、如何使用 Java 的反射机制以及如何对系统进行分层等知识点。

### 9.2 建立界面

在实现功能前，我们需要为这个系统建立界面和设计数据库，本小节将建立在这个小系统中所需要的各个界面。

#### 9.2.1 登录界面

在进入系统前，我们需要经过简单的认证才能进入，因此需要提供一个简单的登录界面，当输入用户名和密码都正确时，就可以进入系统。需要建立的登录界面如图 9.1 所示。

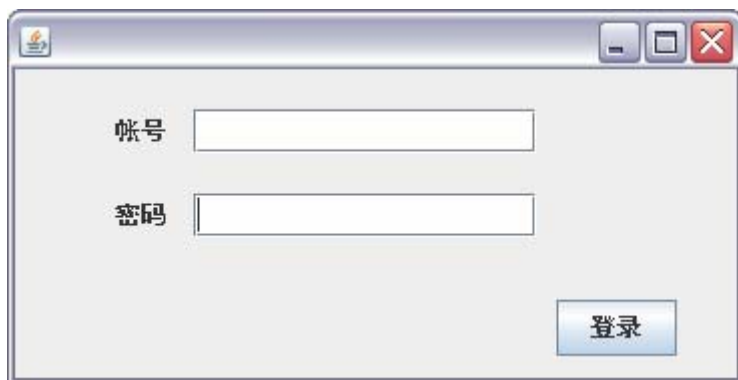


图 9.1 登录界面

界面非常的简单，只是两个输入框，一个按钮即可实现。帐号输入框使用的是 `JTextField` 类，而密码输入框使用的是 `JPasswordField` 类，使用 `JPasswordField` 对象并不会显示原始输入的字符。在本章中，我们将在最后再去实现用户的登录，先去实现系统的销售和入库功能。

#### 9.2.2 销售管理界面

销售管理界面主要用于进行书的销售操作，添加一本需要交易的书、从交易列表中移除该书、进行交易等操作，在本界面的上半部分为交易记录列表，我们约定交易记录列表所显示的为今天进行的交易，并提供一个输入日期的输入框，可以根据日期进行查询该天的交易记录。本界面的下半部分为显示某笔交易的具体信息，包括该笔交易所涉及的金额、销售量、交易日期、交易所涉及的书等信息。初步的界面如图 9.2 所示。

图 9.2 销售界面

如图 9.2，销售管理界面的交易记录列表中，每列的字段分别为该交易所涉及的书本名、总价、交易日期、总数量。销售管理界面的下面部分，是显示具体某笔交易的界面，显示对应某笔交易的总价、交易日期、总数量、交易涉及书的列表，其中交易中涉及书的列表中，包括书的书名、单价、和本次交易中该本书出售的数量。交易中涉及的书列表下，可以选择书和输入书的数量，并提供添加和删除按钮，当选择了一本书并输入相应的交易数量，再点击添加时，即可向交易涉及的书列表中添加书的记录，当然也可以进行删除操作。在界面的最上面，有一个查询按键，可以输入日期进行交易记录查询。

注：具体某笔交易的界面（界面的下面部分），总价、交易日期和总数量是不可输入的，这是由于总价是由各本书的单价乘以交易数量得出来的，交易日期是当前交易时系统的时间为准的，总数量是各本书的交易数量总和，因此并不需要人手进行输入。

另外，如果在书本的下拉框中选择了某本书时，将会带出该本书的单价和库存，好让使用者在操作时对该本书的状况有所了解。

在本例中，每一个界面对应一个 `JPanel`，每个界面都是一个 `JPanel` 类的子类，销售管理界面里面是一个 `JSplitPane` 对象，上面放一个个 `JScrollPane` 对象，下面部分也是一个 `JScrollPane` 对象。上面的 `JScrollPane` 对象主要存放一个 `JTable`，下面的 `JScrollPane` 主要放一些 `Box` 对象进行布局，详细请看图书进存销系统代码清单中的 `SalePanel` 类。

### 9.2.3 入库管理界面

入库管理界面和销售管理界面的布局大致相同，只是其中的数据列和文字有所差距。入加管理界面如图 9.3 所示。

图 9.3 库存管理界面

与销售管理界面类似，上面的列表是入库记录，一条入库记录包括入库时所涉及的书本、入库日期和入库书本的总数量。界面下面部分是具体某条入库记录的具体信息，包括入库日期、总数量和入库书本的列表，同样地，入库日期与总数量都不可以手动输入。在新增一条入库记录时，可以向书本列表添加和删除书本，表示该入库记录中所涉及的书本及对应的数量。

注：在选择某本书时，需要自动带出该书的库存。

在本例中，入库管理界面对应的类为 `RepertoryPanel` 类，是 `JPanel` 的子类。

#### 9.2.4 书本管理界面

书本管理界面主要包括书本的查询、列表、查看等功能，书本在本例中作为基础数据的角色，主要用于销售和入库。这里需要注意的是，在新增一本书的时候，该书的库存为 `0`，只有经过销售和入库才会对书的库存量产生影响。书的基本信息除了书名、价格介绍、所属出版社和书的种类外，还需要有书的图片，为简单起见，本例中的每本书只有一张图片，因此不需要另外建立表来存放书的图片。书本管理界面如图 9.4 所示。



图书进存销管理系统

系统

书名:

书本名称	简介	作者	所属种类	出版社	库存	价格
疯狂Java讲义	疯狂Java讲义	李刚	计算机种类	电子工业出版社	10	50

书本名称:  价格:  作者:

所属种类:  出版社:  书本图片:

书本简介:

图 9.4 书本管理界面

书本管理界面上面的列表主要显示书本名称、简介、所属种类、出版社、库存数量和价格这些信息，此处的书本名称与销售管理（入库管理）界面中的书本名称有所区别，这里的书本名称的列宽较窄，这是由于销售（入库）中所涉及的是多本书，而书本管理界面中每一条书的记录只是代表一本书。

在界面的下方有一个表单，用于查看、修改和添加书本操作，表单的右边是书的图片显示区，用于显示书本所对应的缩略图，当用户点击缩略图的时候，可以弹出新的窗口用于展示大图。

与前面的界面一样，在程序中，我们新增一个 **BookPanel** 作为该界面。

### 9.2.5 种类管理界面

书的种类，表示一本书是属于哪个种类的，例如计算机、文学、法律等等，这些都是书的种类，我们提供一个种类管理界面，可以用于管理各个种类。种类在本例中与书本一样，属于基础数据。新增种类并不需要任何约束，只需要输入种类的名称即可。在书的管理界面中，如果需要添加一本书，必须选择该书所对应的种类。种类管理界面如图 9.5 所示。

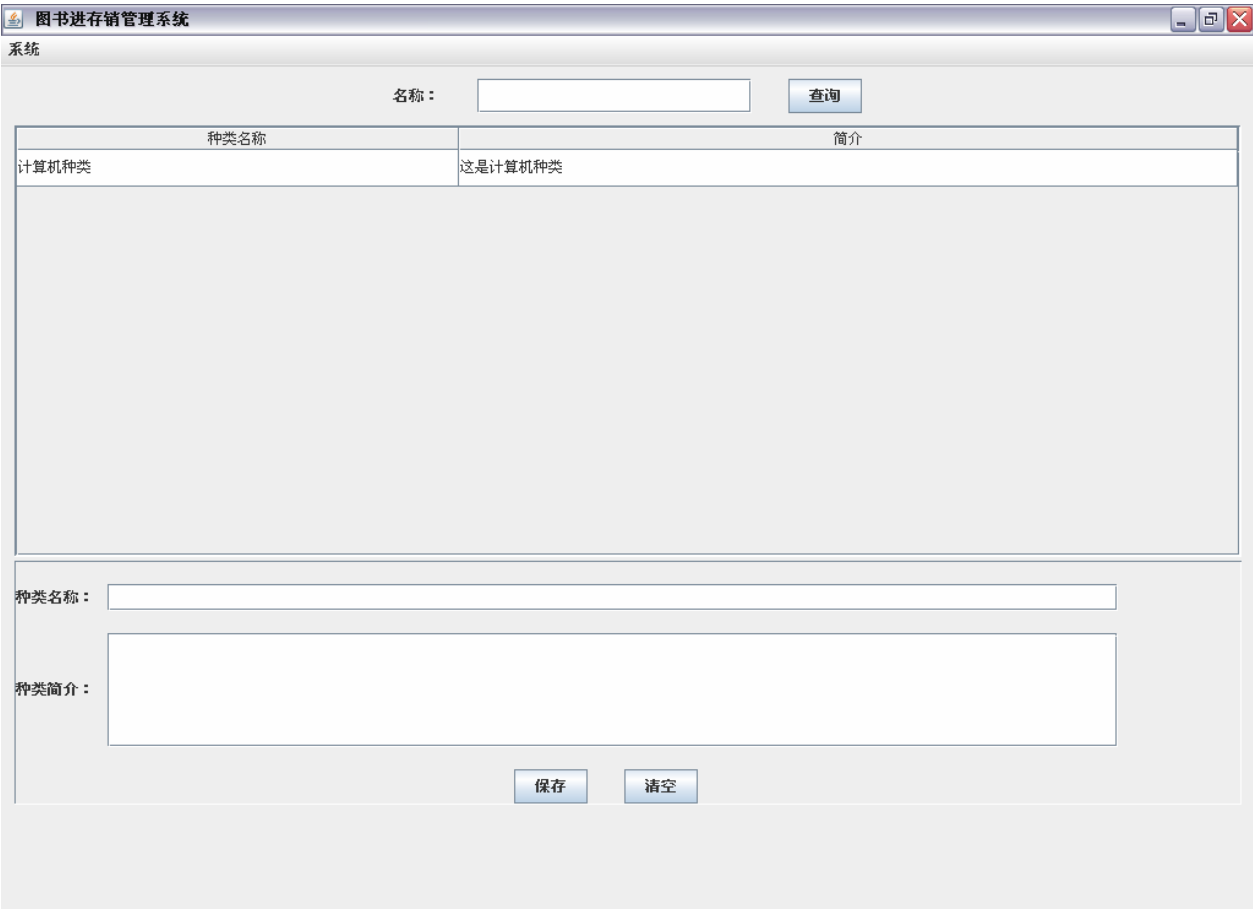


图 9.5 种类管理界面

如图 9.5 所示，种类管理界面并不复杂，上面是一个简单的列表，下面是普通的查看、新增与修改的表单。在程序中，我们新增一个 `TypePanel` 作为该界面。

9.2.6 出版社管理界面

添加一本书，除了需要有种类之外，还需要选择该书的出版社，我们提供一个出版社管理界面。如图 9.6 所示。

图 9.6 出版社管理界面

出版社列表中，主要显示出版社名称、联系人、联系电话和简介这些信息，下面的表单与前面几个界面的表单类似，都是用于查看、修改和新增。在程序中，我们新增一个 **ConcernPanel** 来表示这个界面。

从 9.2.2 到 9.2.6 的各个系统界面，都是存在于一个 **JFrame** 中，我们可以提供菜单，当点击了某个菜单的时候，**JFrame** 中的各个 **JPanel** 对象隐藏，只显示对应的那个 **JPanel**。

到此，我们所需要的界面已经全部完成了，在 9.2.7 中我们将对各个界面的代码进行优化，再去为主界面加上相应的菜单，我们的界面就全部完成。

## 9.2.7 修改界面代码

本例中涉及各个界面，我们可以发现界面几乎都大同小异，界面上面部分是一个列表，下面是一个表单，因此，我们可以将这几个界面的共同部分提取出来，作为每个界面对象（**JPanel**）的父类，将一些可以重用的代码提升至父类，并可以使用“模板方法”，提供一些抽象的方法让各个子类去实现。

以下代码为各个界面对象的父类（**CommonPanel**）的代码。

代码清单：code\book\src\org\crazyit\book\ui\CommonPanel.java

```
private JTable table; //存放数据的 table
protected Vector<Vector> datas; //列表数据
public void setJTable(JTable table) {
    this.table = table;
}
public JTable getJTable() {
```

```

        return this.table;
    }
    public Vector<Vector> getDatas() {
        return datas;
    }
    public void setDatas(Vector<Vector> datas) {
        this.datas = datas;
    }
    //将数据设置进 JTable 中
    public void initData() {
        if (this.table == null) return;
        DefaultTableModel tableModel = (DefaultTableModel)this.table.getModel();
        //将数据设入表格 Model 中
        tableModel.setDataVector(getDatas(), getColumns());
        //设置表格样式
        setTableFace();
    }
    //刷新列表的方法
    public void refreshTable() {
        initData();
        getJTable().repaint();
    }
    //获取表列集合, 由子类去实现
    public abstract Vector<String> getColumns();
    //设置列表的样式, 由子类去实现
    public abstract void setTableFace();
    //设置数据列表的方法, 由子类去实现
    public abstract void setViewDatas();
    //清空界面下边的列表
    public abstract void clear();
    //给子类使用的方法, 用于获取一个列表的 id 列值
    public String getSelectId(JTable table) {
        int row = table.getSelectedRow();
        int column = table.getColumn("id").getModelIndex();
        String id = (String)table.getValueAt(row, column);
        return id;
    }
}

```

以上的代码, 提供一个 **JTable** 的属性, 这是因为每个界面中都有一个主要的列表对象, 例如销售管理界面中的销售记录列表、书本管理界面中的书本列表等。注意代码中的黑体部分, 都是由子类去实现的方法, **getColumns()** 由子类去提供列表的列集合; **setTableFace()** 是由子类去设置 **JTable** 的显示, 例如需要设置某一列的宽度或者设置列表的行高等等; 而 **setViewDatas()** 方法是重新去数据库读取数据, 由子类去实现; **clear()** 方法是刷新每个界面下边的表单, 同样由子类去实现。在 **CommonPanel** 中提供了一个 **getSelectId** 方法, 用来获得 **JTable** 属性中所选中的某一行中 **id** 列的值, 也就是意味着在列表中, 必须要有一个列名叫做 **id** 的列。

父类定义好规范之后, 子类就可以根据不同的情况, 给父类不同的列, 让父类进行显示, 除了需要提供列的集合外, 还需要提供数据, 也就是 **CommonPanel** 中的 **datas** 属性。我们这里提供了一个 **setViewDatas** 的方法, 主要从数据库中读取相关的数据, 再调用父类的 **initDats** 方法构建列表。那么存放各个界面对象的 **JFrame** 类中, 可以提供一个方法, 当界面发生转换时, 就调用 **setViewDatas** 方法, 具体代码如下。

代码清单: code\book\src\org\crazyit\book\ui\MainFrame.java

```
//切换各个界面
private void changePanel(CommonPanel commonPanel) {
    //移除当前显示的 JPanel
    this.remove(currentPanel);
    //添加需要显示的 JPanel
    this.add(commonPanel);
    //设置当前的 JPanel
    this.currentPanel = commonPanel;
    this.repaint();
    this.setVisible(true);
    //调用 CommonPanel 的方法重新读取数据并刷新列表
    commonPanel.setViewDatas ();
    //清空界面下边的表单
    commonPanel.clear();
}
```

以上代码的黑体部分,调各个界面对象父类的 `setViewDatas` 方法重新读取数据并刷新列表,各个界面中转换时,当点击了对应的菜单后,再进行转换(调用 `changePanel` 方法)。

代码清单: code\book\src\org\crazyit\book\ui\MainFrame.java

```
private Action sale = new AbstractAction("销售管理", new ImageIcon("images/sale.gif")) {
    public void actionPerformed(ActionEvent e) {
        //调用转换的方法
        changePanel(salePanel);
    }
};
```

以上代码表示点击了销售管理的菜单后,就调用 `changePanel` 方法,转换界面并初始化数据。

另外,每个界面的主列表我们使用一个 `CommonJTable` 对象,该对象继承于 `JTable`,我们并不需要让列表的每个单元格可以编辑,因此重写 `JTable` 的 `isCellEditable` 方法即可,代码如下。

代码清单: code\book\src\org\crazyit\book\ui\CommonJTable.java

```
public class CommonJTable extends JTable {
    public CommonJTable(TableModel dm) {
        super(dm);
        //设置表格只能选择一行
        getSelectionModel().setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    }
    //重写父类的方法,使所有的单元格不可编辑
    public boolean isCellEditable(int row, int column) {
        return false;
    }
}
```

做了以上的准备工作后,就可以修改各个界面对应的类,去继承 `CommonPanel` 即可,实现 `getColumns()`、`setTableFace()`、`setViewDatas()`和 `clear()`方法,而数据暂时不必提供,下一节我们将开始设计数据库。

## 9.3 设计系统数据库

在 9.2 中我们已经建立了系统的相关界面,接下来,这一小节将设计这个系统的数据库。本例使用的是 `MySQL5.0` 作为数据库。在设计数据库前,我们可以确定,系统相关的表,从最基础开始,有出版

社表、书的类型表、书表、入库记录表和销售记录表，其中，一条入库记录中涉及多本书，一条销售记录也涉及多本书，因此还需要书的入库记录表和书的销售记录表，一条书的入库记录对应一本书，该条书的入库记录属于某一条入库记录，可以理解成这是书与入库记录的关系表，同样地，书的销售记录与书的入库记录一样。

如果上面的文字难以理解，可以看下面的数据库结构，更有助于理解。在设计各个表前，我们需要创建数据库，在 MySQL 中，创建 **BOOK\_SYSTEM** 数据库，具体的 SQL 语句如下：

```
-- 创建 DATABASE
CREATE DATABASE IF NOT EXISTS BOOK_SYSTEM;
-- 使用 BOOK_SYSTEM
USE BOOK_SYSTEM;
```

MySQL 中的 CREATE 和 USE 语法可查看 MySQL 的帮助文档。

### 9.3.1 设计出版社表

在 9.2.6 中，我们已经建立了出版社的管理界面，现在只需要根据界面来设计数据库即可。我们在 MySQL 中建立表 **T\_PUBLISHER**，出版社表包括的字段有：

- ❑ ID：主键 ID。
- ❑ PUB\_NAME：出版社名称。
- ❑ PUB\_TEL：联系电话。
- ❑ PUB\_LINK\_MAN：联系人。
- ❑ PUB\_INTRO：出版社简介。

由于出版社是最基础的数据，因此不需要其他的外键关系。以下为创建 **T\_PUBLISHER** 这个表的 SQL。

```
-- 出版社
CREATE TABLE IF NOT EXISTS `T_PUBLISHER` (
  `ID` int AUTO_INCREMENT NOT NULL, -- 主键生成策略为自动增长
  `PUB_NAME` varchar(50), -- 出版社名称
  `PUB_TEL` varchar(50), -- 联系电话
  `PUB_LINK_MAN` varchar(50), -- 联系人
  `PUB_INTRO` varchar(200), -- 简介
  PRIMARY KEY (`ID`) -- 声明主键
);
```

如果你有 MySQL 的一些管理工具，可以使用这些工具进行创建表，并不需要自己书写 SQL 语句，但笔者还是希望能够自己尝试去编写这些 SQL，因为对于初学者而言，可以自己编写这些 SQL，无疑是对自己能力的一种提高。

### 9.3.2 设计种类表

种类表与出版社表一样，并不复杂，也没有任何的外键，根据 9.2.5 的界面，就可以确定需要哪些字段了。创建 **T\_BOOK\_TYPE** 表，包括以下的字段：

- ❑ ID：主键 ID
- ❑ TYPE\_NAME：种类的名称。
- ❑ TYPE\_INTRO：种类的简介。

只有简单的三个字段，创建的 SQL 如下。

```
-- 书种类
CREATE TABLE IF NOT EXISTS `T_BOOK_TYPE` (
```

```

`ID` int AUTO_INCREMENT NOT NULL, -- 主键生成策略为自动增长
`TYPE_NAME` varchar(50), -- 种类名称
`TYPE_INTRO` varchar(200), -- 种类简介
PRIMARY KEY (`ID`)
);

```

### 9.3.3 设计书表

在 9.2.4 的界面中，我们可以看到，一本书所需要的字段包括：书名、简介、作者、所属种类、出版社、缩略图地址、库存和价格，种类表和出版社表已经在 9.3.2 与 9.3.1 中设计了，只需要为书表添加外键关系即可，新增表 **T\_BOOK**，以下为书表各个字段的描述：

- ❑ **ID**：主键 ID
- ❑ **BOOK\_NAME**：书的名字。
- ❑ **BOOK\_INTRO**：书的简介。
- ❑ **BOOK\_PRICE**：书的单价。
- ❑ **TYPE\_ID\_FK**：书所属的种类 ID 外键。
- ❑ **PUB\_ID\_FK**：出版社外键。
- ❑ **IMAGE\_URL**：缩略图的地址。
- ❑ **AUTHOR**：作者名称。
- ❑ **REPERTORY\_SIZE**：库存量。

这里需要注意一下，书的名字、简介、单价、所属的种类、出版社、缩略图和作者，都可以在界面中通过输入或者选择写入到数据库中，但是库存量是由销售与入库决定的，因此在修改书的时候，不可以设置书的库存量。创建该表的 SQL 如下。

```

-- 书
CREATE TABLE IF NOT EXISTS `T_BOOK` (
  `ID` int AUTO_INCREMENT NOT NULL, -- ID 字段，自增
  `BOOK_NAME` varchar(50), -- 书名称
  `BOOK_INTRO` varchar(200), -- 书简介
  `BOOK_PRICE` double, -- 书的单价
  `TYPE_ID_FK` int NOT NULL, -- 种类外键
  `PUB_ID_FK` int NOT NULL, -- 出版社外键
  `IMAGE_URL` varchar(200), -- 缩略图 URL
  `AUTHOR` varchar(200), -- 作者
  `REPERTORY_SIZE` bigint(10), -- 库存数量
  FOREIGN KEY (`TYPE_ID_FK`) REFERENCES `T_BOOK_TYPE` (`ID`), -- 声明种类的外键
  FOREIGN KEY (`PUB_ID_FK`) REFERENCES `T_PUBLISHER` (`ID`), -- 声明出版社外键
  PRIMARY KEY (`ID`)
);

```

### 9.3.4 设计入库记录表

在设计入库记录前，我们需要知道的是，一个入库记录，包含多本书的入库，也就是说，一次入库，有可能入库多本书。入库记录，只需要记录入库的时间即可，新建 **T\_IN\_RECORD** 表，该表的字段如下：

- ❑ **ID**：主键 ID。
- ❑ **RECORD\_DATE**：入库日期。



以下为创建该表的 SQL。

```
-- 入库记录表，一次入库会入多本书
CREATE TABLE IF NOT EXISTS `T_IN_RECORD` (
  `ID` int AUTO_INCREMENT NOT NULL,
  `RECORD_DATE` datetime, -- 入库日期
  PRIMARY KEY (`ID`)
);
```

设计完入库记录后，我们还需要考虑，一次入库涉及了多本书，因此需要添加一个关系表，用来表示一次入库中所涉及的书。创建 `T_BOOK_IN_RECORD` 表，该表的字段如下：

- ❑ `ID`：该表的主键 `ID`
- ❑ `BOOK_ID_FK`：书的外键，表示这一条书的入库记录所对应的书。
- ❑ `T_IN_RECORD_ID_FK`：入库记录的外键，表示这一条书的入库记录所对应的是哪一次入库，这个关系也表示了一次入库可以有多条书的入库记录。
- ❑ `IN_SUM`：这一条书的入库记录中对应的书的入库数量。

创建的 SQL 语句如下。

```
-- 书的入库记录
CREATE TABLE IF NOT EXISTS `T_BOOK_IN_RECORD` (
  `ID` int AUTO_INCREMENT NOT NULL, -- ID 自增
  `BOOK_ID_FK` int, -- 入库的书
  `T_IN_RECORD_ID_FK` int, -- 对应的入库记录
  `IN_SUM` int(10), -- 入库数量
  FOREIGN KEY (`BOOK_ID_FK`) REFERENCES `T_BOOK` (`ID`), -- 声明书的外键
  FOREIGN KEY (`T_IN_RECORD_ID_FK`) REFERENCES `T_IN_RECORD` (`ID`), -- 声明入库记录外键
  PRIMARY KEY (`ID`)
);
```

注：在入库表中，并没有提供入库总数量这个字段，这是由于入库总数量是各本书的入库数量的总和。因此并不需要在数据库中提供字段，只需要在程序中进行计算即可。

创建完了这两个表之后，就完成了书的入库相关表的设计。

### 9.3.5 设计销售记录表

销售记录表与入库记录表大同小异，都是需要创建一个销售表，再创建书的销售表。一条销售记录对应多条书的销售记录，一次销售所涉及多本书，与入库记录一样，都通过一个中关系表来体现这种关系。销售记录表的字段如下：

- ❑ `ID`：`ID` 主键。
- ❑ `RECORD_DATE`：交易日期。

创建该表的 SQL 如下。

```
-- 交易记录表，一个交易记录包括多个书的销售记录，一次交易可能有多本书
CREATE TABLE IF NOT EXISTS `T_SALE_RECORD` (
  `ID` int AUTO_INCREMENT NOT NULL,
  `RECORD_DATE` datetime, -- 交易日期
  PRIMARY KEY (`ID`)
);
```

创建了交易表后，再去设计书的交易表，具体字段与书的入库记录表相似：

- ❑ `ID`：主键 `ID`
- ❑ `BOOK_ID_FK`：该条书的交易记录所对应的书。

❑ **T\_SALE\_RECORD\_ID\_FK**: 该条书的交易记录所对应的交易记录。

❑ **TRADE\_SUM**: 该记录中对应的书的交易数量。

创建书的入库记录表的 SQL 如下。

```
-- 书的销售记录, 一条记录对应一本书
CREATE TABLE IF NOT EXISTS `T_BOOK_SALE_RECORD` (
  `ID` int AUTO_INCREMENT NOT NULL,
  `BOOK_ID_FK` int, -- 销售的书
  `T_SALE_RECORD_ID_FK` int, -- 该书的销售记录所对应的交易记录
  `TRADE_SUM` int(10), -- 销售数量
  FOREIGN KEY (`BOOK_ID_FK`) REFERENCES `T_BOOK` (`ID`),
  FOREIGN KEY (`T_SALE_RECORD_ID_FK`) REFERENCES `T_SALE_RECORD` (`ID`),
  PRIMARY KEY (`ID`)
);
```

注: 交易表中并没有提供交易总价和交易总数量这两个字段, 这是由于这两个值是由各本书的交易量所决定的。

到此, 数据库各个表的结构已经设计完成了, 各个表的结构都十分简单, 只有库存与销售两个模块的表相对复杂了一点。在下一节中, 我们将讲解如何使用 Java 的反射进行对象与表的映射与使用 JDBC 进行数据库操作等知识。

## 9.4 开发前的准备工作

在 9.3 小节中, 数据库已经设计好了, 接下来可以准备开发的工作了。其实当设计好数据库, 我们就可以进行开发, 本小节中所讲的开发准备工作, 是指编写一些公用的方法, 例如 JDBC 的基本操作, 包括查询、修改等。将一些可以重用的代码先编写好, 再去进行业务开发, 到业务开发的时候, 可以事半功倍。

### 9.4.1 设计表的对应类

在 9.3 中, 已经设计了系统所需要的各个表, 包括出版社表、书种类表、书表等, 那么现在, 我们需要为这些表建立相应的对象, 每一个表可以对应一个对象。可以为每个表的对象先建立一个父类, 由于每个表中都有 ID 一列, 因此我们可以建立一个父类, 提供 ID 字段。

新建各个表对应对象的父类 ValueObject, ValueObject 的代码如下。

代码清单: code\book\src\org\crazyit\book\vo\ValueObject.java

```
public class ValueObject {
    //ID 字段, 对应数据库中的 ID 列
    private String ID;
    //省略 setter 和 getter 方法
}
```

该类只提供了一个 ID 字段, 并提供 setter 和 getter 方法, 注意, ID 属性必须与数据库中表的 ID 列名字对应, 由于这个是个表对象的父类, 因此需要约定每个表的主键命名必须为 ID, 为什么需要这样, 在下面章节将会详细讲述。

建立了父类 ValueObject 后, 此时再去新建出版社所对应的实体 Concern 类, 一个 Concern 对象代表一个出版社, 该类继承于 ValueObject, 此外, 出版社表中有的字段, 都需要在这个类中反应出来。Concern 类代码如下。

代码清单: code\book\src\org\crazyit\book\vo\Concern.java

```
public class Concern extends ValueObject {
```

```

//出版社名称
private String PUB_NAME;
//出版社电话
private String PUB_TEL;
//联系人
private String PUB_LINK_MAN;
//简介
private String PUB_INTRO;
//省略各个属性的 getter 和 setter 方法
}

```

注意各个属性的命名，必须要与数据库中表的字段一致，由于继承了 `ValueObject` 类，因此不需要提供 ID 字段。按照建立 `Concern` 对象的方法，再去建立 `Type` 对象、`Book` 对象，`Type` 代表书本类，`Book` 代表书类型。在 `Book` 类中需要注意的是，由于书表中有两个键，分别是种类的外键 `TYPE_ID_FK` 和出版社的外键 `PUB_ID_FK`，同样地也提供这两个类属性，并不是提供种类的对象（`Type`）和出版社的对象（`Concern`），另外所有属性类型都需要为 `String`。

`Type` 的代码如下，代码清单：code\book\src\org\crazyit\book\vo\Type.java

```

public class Type extends ValueObject {
    //名称
    private String TYPE_NAME;
    //简介
    private String TYPE_INTRO;
    //省略 setter 和 getter 方法
}

```

`Book` 的代码如下，代码清单：code\book\src\org\crazyit\book\vo\Book.java

```

public class Book extends ValueObject {
    private String BOOK_NAME; //书本名称
    private String BOOK_INTRO; //简介
    private String BOOK_PRICE; //书的单价
    private String TYPE_ID_FK; //种类外键
    private String PUB_ID_FK; //出版社外键
    private String REPERTORY_SIZE; //存储量
    private String IMAGE_URL; //图片 url
}

```

创建完三个基础数据表所对应的类后，接下来再去创建入库记录表对应的类、书的入库记录表对应的类、销售记录表对应的类和书的销售记录表对应的类。

入库记录类 `InRecord`，代码清单：code\book\src\org\crazyit\book\vo\InRecord.java

```

public class InRecord extends ValueObject {
    private String RECORD_DATE; //入库日期
    //省略 getter 和 setter 方法
}

```

书的入库记录类 `BookInRecord`，代码清单：code\book\src\org\crazyit\book\vo\BookInRecord.java

```

public class BookInRecord extends ValueObject {
    private String BOOK_ID_FK; //对应书的外键，从数据库查出来时有值
    private String T_IN_RECORD_ID_FK; //对应销售记录外键
    private String IN_SUM; //入库数量
    //省略 setter 和 getter 方法
}

```

销售记录类 `SaleRecord`，代码清单：code\book\src\org\crazyit\book\vo\SaleRecord.java

```

public class SaleRecord extends ValueObject {

```

```
private String RECORD_DATE; //交易日期
//省略 setter 和 getter 方法
}
```

书的销售记录类 `BookSaleRecord`。

代码清单：`code\book\src\org\crazyit\book\vo\BookSaleRecord.java`

```
public class BookSaleRecord extends ValueObject {
    private String BOOK_ID_FK; //该记录对应的书的外键
    private String T_SALE_RECORD_ID_FK; //该记录对应的销售记录的外键
    private String TRADE_SUM; //该记录所对应的书的销售数量
    //省略 setter 和 getter 方法
}
```

到现在，与系统相关的各个表所对应的类都已经编写好了，下面小节，我们将讲解这些建立好的类在开发过程中所体现的作用。

### 9.4.2 编写配置读取类

由于本章涉及到数据库操作，因此与数据库相关的一些配置，例如对应数据库的相关驱动、数据库地址、用户名和密码，我们可以放到配置文件中，如果需要更换数据库或者地址，只需要修改这份配置文件即可。

建立配置文件 `jdbc.properties`，内容如下：

```
//JDBC 驱动
jdbc.driver=com.mysql.jdbc.Driver
//连接地址
jdbc.url=jdbc:mysql://localhost:3306/book_system
//数据库用户名
jdbc.user=book
//密码
jdbc.pass=book
```

建立好该文件后，再编写类去读取该文件，获得所需要的值即可。用于读取配置的 `PropertiesUtil` 类，代码如下。

代码清单：`code\book\src\org\crazyit\book\jdbc\PropertiesUtil.java`

```
public class PropertiesUtil {
    private static Properties properties = new Properties(); //该记录所对应的书的销售数量
    private static String CONFIG = "cfg/jdbc.properties"; //配置文件的路径
    //读取资源文件，设置输入流
    private static InputStream is = PropertiesUtil.class.getResourceAsStream(CONFIG);
    public static String JDBC_DRIVER; //数据库驱动
    public static String JDBC_URL; //jdbc 连接 url
    public static String JDBC_USER; //数据库用户名
    public static String JDBC_PASS; //数据库密码
    static {
        properties.load(is); //加载输入流
        //获得配置各个属性
        JDBC_DRIVER = properties.getProperty("jdbc.driver");
        JDBC_URL = properties.getProperty("jdbc.url");
        JDBC_USER = properties.getProperty("jdbc.user");
        JDBC_PASS = properties.getProperty("jdbc.pass");
    }
}
```

读取的各个配置的作用，将在 9.4.3 中作详细讲解。

### 9.4.3 编写JDBC操作类

JDBC 是 Java Data Base Connectivity 的简称，是 Java 中进行数据库连接的技术。JDBC 的 API 提供了标准统一的 SQL 数据存取接口，可以让程序员不需要关心如何去连接不同的数据库，只需为不同的数据库提供不同的驱动，就可以达到连接不同数据库的要求。

在 9.4.2 中，我们已经提供了配置，可以修改对应的配置文件来连接数据库，jdbc.properties文件中的jdbc.driver属性，就是数据库的连接驱动，在本例中我们使用了MySQL数据库，因此需要提供MySQL的数据库驱动包。本例中使用的驱动包版本为 5.1.6，如果需要最新的驱动程序，请到<http://dev.mysql.com/downloads/connector/j/5.1.html>下载，下载后将驱动包加到环境变量中。除了配置驱动外，还需要配置数据库的连接地址、用户名和密码，jdbc.url=jdbc:mysql://服务器ip:3306/book\_system，用户名密码为你的MySQL用户密码。

配置好了之后，我们可以开始着手编写数据库的操作类，但在那之前，我们需要明确这个类帮我做些什么。首先肯定是帮我们进行数据库连接，我们之前配置了连接的相关属性，但是程序并不知道我们需要怎样去连接，接着我们需要这个类帮我们提供查询、执行 SQL 等功能。确定好目标后，开始编写。

新建 JDBCExecutor 类，该类具有属性如下：

```
private static String DRIVER = PropertiesUtil.JDBC_DRIVER; //获得驱动
private static String URL = PropertiesUtil.JDBC_URL; //获得 url
private static String USER = PropertiesUtil.JDBC_USER; //获得连接数据库的用户名
private static String PASS = PropertiesUtil.JDBC_PASS; //获得连接数据库的密码
private Connection connection; //连接对象
private static JDBCExecutor jdbcExecutor; //维护一个本类型的对象
private Statement stmt; //维护一个本类型的对象
```

注：以上代码的黑体部分，由于创建一个 Connection 对象需要耗费很大的资源，因此我们使用单态模式，让 JDBCExecutor 类维护一个 JDBCExecutor 对象，可以在构造器中创建 Connection，由于 JDBCExecutor 是单态的，因此可以保证在应用中只创建一个 Connection，单态模式将在下一小节中详细讲述。

下面在 JDBCExecutor 的构造器中创建各个对象，再提供一个方法返回 JDBCExecutor 的实例。

代码清单：code\book\src\org\crazyit\book\jdbc\JDBCExecutor.java

```
//私有构造器
private JDBCExecutor() {
    //初始化 JDBC 驱动并让驱动加载到 jvm 中
    Class.forName(DRIVER);
    //创建数据库连接
    connection = DriverManager.getConnection(URL, USER, PASS);
    //创建 Statement 对象
    stmt = connection.createStatement();
}
//提供一个静态方法返回本类的实例
public static JDBCExecutor getJDBCExecutor() {
    //如果本类所维护 jdbcExecutor 属性为空,则调用私有的构造器获得实例
    if (jdbcExecutor == null) jdbcExecutor = new JDBCExecutor();
    return jdbcExecutor;
}
```

注：在以上代码中，提供了一个 JDBCExecutor 的私有构造器，因为需要保持这个类只创建一次，因此不可提供 public 的构造器让其他类去创建 JDBCExecutor 的实例，外界只能通过它自己内部的一个静态方法创建 JDBCExecutor 实例。

编写执行查询的方法，代码清单：code\book\src\org\crazyit\book\jdbc\JDBCExecutor.java

```
//执行一句查询的 sql, 并返回 ResultSet 对象
public ResultSet executeQuery(String sql) {
    //利用 Statement 对象执行参数的 sql
    ResultSet result = stmt.executeQuery(sql);
    return result;
}
```

在上面的代码中，并没有关闭 ResultSet 的代码，直接返回 ResultSet 对象，我们还需要对 ResultSet 对象进行一些处理，因此在这里不进行关闭操作，该方法只是简单的进行查询。

编写执行 SQL 的方法，代码清单：code\book\src\org\crazyit\book\jdbc\JDBCExecutor.java

```
//执行单句 INSERT、UPDATE 或 DELETE 语句，如果执行 INSERT 时，返回主键
public int executeUpdate(String sql) {
    int result = -1;
    //执行 SQL 语句
    stmt.executeUpdate(sql);
    //获得主键
    ResultSet rs = stmt.getGeneratedKeys();
    while(rs.next()) result = rs.getInt(1); //返回最后一个主键
    rs.close();
    return result;
}
```

在以上代码中，将会返回执行该句 SQL 所产生的主键。到这里，我们的 JDBCExecutor 类已经编写完了。

#### 9.4.4 创建数据转换工具类

在 9.4.3 中，JDBCExecutor 中提供了一个 executeQuery 方法，该方法返回 ResultSet 对象，当时我们并没有关闭 ResultSet，这是由于我们需要对该结果集进行一些封装。在本小节中将建立一个工具类，进该结果集进行封装，并返回对应的集合。

在 9.4.1 中，我们设计了各个表所对应的类，那么在封装的过程中，将结果集（ResultSet）中的某个值作为这些类（表对象）的属性，并负责创建这些对象，放到集合中去。新建 DataUtil 类，具体的代码如下。

代码清单：code\book\src\org\crazyit\book\commons\DataUtil.java

```
//将 rs 中的值封装成一个集合
public static Collection getDatas(Collection result, ResultSet rs, Class clazz) {
    while (rs.next()) {
        //创建类的实例
        Object vo = clazz.newInstance();
        //获取本对象的属性
        Field[] fields = clazz.getDeclaredFields();
        //获取父类的属性
        Field[] superFields = clazz.getSuperclass().getDeclaredFields();
        //父类的属性和自己的属性相加
        Field[] allFields = addFields(superFields, fields);
        //遍历所有的属性
```

```

        for (Field field : allFields) {
            //获得 setter 方法的方法名
            String setterMethodName = getSetterMethodName(field.getName());
            //获得 setter 方法
            Method setterMethod = clazz.getMethod(setterMethodName, field.getType());
            invokeMethod(rs, field, vo, setterMethod);
        }
        result.add(vo);
    }
    rs.close();
    return result;
}

//执行一个方法，从 ResultSet 中获取一个字段的值，调用 vo 的 setter 方法
private static void invokeMethod(ResultSet rs, Field field, Object vo,
    Method setterMethod) {
    //当使用 ResultSet 获取某个字段的时候，如果没有该字段，会出现 SQLException，在这里忽略该异常
    String value = rs.getString(field.getName());
    //从 ResultSet 中获取与该对象属性名一致的字段，并执行 setter 方法
    setterMethod.invoke(vo, value);
}

//根据属性名获得 setter 方法的方法名
private static String getSetterMethodName(String fieldName) {
    String begin = fieldName.substring(0, 1).toUpperCase();
    String end = fieldName.substring(1, fieldName.length());
    String methodName = "set" + begin + end;
    return methodName;
}

//相加两个数组
private static Field[] addFields(Field[] f1, Field[] f2) {
    List<Field> l = new ArrayList<Field>();
    for (Field f : f1) l.add(f);
    for (Field f : f2) l.add(f);
    return l.toArray(new Field[f1.length + f2.length]);
}
}

```

注意以上代码的黑体部分，先使用 `class` 的 `newInstance` 方法创建类的实例，再获得父类和本类的属性，再通过 `setter` 方法将 `ResultSet` 中对应的值设置到对象中。我们在 9.4.1 中，约定了各个类的属性命名都需要与数据库中的表字段一致，在这里，我们很容易就可以得到某个字段的 `setter` 方法，并可以根据类的属性名称得到 `ResultSet` 中对应的字段值。在本例中，父类就是 `ValueObject`，`ValueObject` 中有一个 `ID` 属性，就对应了各个表中的 `ID` 字段。`ValueObject` 下面的各个子类的 `class` 都可以作为 `getDatas` 的 `clazz` 参数传入，从而创建传入类的对象。

简单的说，`DataUtil` 就是根据 `ResultSet` 来创建 `ValueObject` 的集合，当创建完后，就需要在这里关闭 `ResultSet` 对象。这也是为什么在 `JDBCExecutor` 不需要关闭 `ResultSet` 的原因。

到这里，开发的准备工作已经全部完成了，在本小节中，描述了如何连接 `JDBC` 进行数据库连接，使用 `Java` 的反射进行字段与类属性的映射，这些都为我们后面的功能开发打好了基础，在开发的过程中，我们可以直接编写 `SQL`，得到结果集，传递给 `DataUtil` 类，让它转换成集合，我们不再需要去处理 `ResultSet`、`Connection` 和 `Statement` 等对象。



## 9.5 出版社管理功能

从本小节开始，我们开始实现系统功能，从最简单的出版社管理功能开始，由于出版社表中不涉及任何的外键，界面中只有一个简单的列表和表单，实现起来比较简单。

### 9.5.1 分层结构

我们可以将系统分为三层：表现层、业务层、数据访问层，这样分层的好处在于，如果视图层发生变化，例如不再使用 **swing** 作为表现层，使用 **jsp** 的话，那么，业务层、数据访问层的代码将不需要改变，达到重用的目的。业务层与数据访问层分别提供各自的接口，在表现层中使用业务层的接口，业务层中使用数据访问层的接口，就算实现发生了改变，也可以不用去更改调用者的代码，当需要更改某一部分实现的时候，直接更换实现类即可。

我们现在开始开发数据访问层的代码，先数据访问层建立一个父类，给该层各个 **DAO** 实现类去继续，提供一些可能每个 **DAO** 都会使用到的方法。新建 **CommonDaolmpl** 对象，具体代码如下。

代码清单：code\book\src\org\crazyit\book\dao\impl\CommonDaolmpl.java

```
public class CommonDaolmpl {
    //返回 JDBCExecutor 对象
    public JDBCExecutor getJDBCExecutor() {
        return JDBCExecutor.getJDBCExecutor();
    }
    //根据参数的 SQL，存放结果的集合对象，和具体的数据库映射对象返回一个集合
    public Collection getDatas(String sql, Collection<ValueObject> result,
        Class clazz) {
        //执行 SQL 返回 ResultSet 对象
        ResultSet rs = getJDBCExecutor().executeQuery(sql);
        //对 ResultSet 进行封装并返回集合
        return DataUtil.getDatas(result, rs, clazz);
    }
}
```

那么在 **DAO** 中每个实现类去继承 **CommonDaolmpl** 的话，即可使用 **getDatas** 方法返回已经封装好的数据集合。

出版社数据访问层接口：

```
public interface ConcernDao {
    //提供需要实现的接口方法
}
```

出版社业务接口：

```
public interface ConcernService {
    //提供需要实现的接口方法
}
```

那么在表现层的 **JPanel** 中，我们可以在 **JPanel** 的构造器中，将 **ConcernService** 作为构造参数传入到 **ConcernPanel**（出版社管理界面）中。同样的，在 **ConcernService** 实现类的构造器中，将 **ConcernDao** 传入。

### 9.5.2 获取全部出版社

新建 **ConcernDao** 的实现类（数据访问层的实现类）**ConcernDaolmpl**，为 **ConcernDao** 添加一个需要实现的方法 **findAll**，以下为 **findAll** 的接口方法及期实现。

ConcernDao, 代码清单: code\book\src\org\crazyit\book\dao\ConcernDao.java

```
public interface ConcernDao {
    //查找全部的出版社并返回集合
    Collection<Concern> findAll();
}
```

ConcernDaoImpl, 代码清单: code\book\src\org\crazyit\book\dao\impl\ConcernDaoImpl.java

```
public class ConcernDaoImpl extends CommonDaoImpl implements ConcernDao {
    public Collection<Concern> findAll() {
        //查找的 SQL
        String sql = "SELECT * FROM T_PUBLISHER pub ORDER BY pub.ID DESC";
        //调用父类的 getDatas 方法返回集合, 返回的类型是集合, 集合里面存放的类型是 Concern
        return getDatas(sql, new Vector(), Concern.class);
    }
}
```

注意以上代码中的黑体部分, 调用父类的方法返回数据集合, 返回怎样的数据集合, 根据参数类型决定。在这里, 就可以体会到在 9.4 中所做的准备工作的重要性, 我们不需要再去处理 **ResultSet** 对象, 即可将结果封装成集合, 减低了代码的耦合。

接下来, 再去实现业务层:

ConcernService, 代码清单: code\book\src\org\crazyit\book\service\ConcernService.java

```
public interface ConcernService {
    //获取全部的出版社
    Collection<Concern> getAll();
}
```

ConcernServiceImpl。

代码清单: code\book\src\org\crazyit\book\service\impl\ConcernServiceImpl.java

```
public class ConcernServiceImpl implements ConcernService {
    //数据访问对象
    private ConcernDao dao;
    //在构造器中设置 ConcernDao
    public ConcernServiceImpl(ConcernDao dao) {
        this.dao = dao;
    }
    //直接返回数据
    public Collection<Concern> getAll() {
        return dao.findAll();
    }
}
```

注: 在业务层中, 并没有对结果集合进行任何处理, 使用数据访问对象得到数据后马上返回, 这是由于我们这里并没有任何复杂的业务, 只需要得到数据即可。

如果需要进行一些业务的处理, 那么可以在业务层的实现类中进行处理。例如, 如果某天有一个需求, 出版社名字中含有英文字母的出版社不显示到表现层中, 那么负责处理这个需求的代码, 就应该写在业务层中。

接下来, 就是表现层如何得到数据并进行显示了。

代码清单: code\book\src\org\crazyit\book\ui\ConcernPanel.java

```
//将数据转换成视图表格的格式
private Vector<Vector> changeDatas(Vector<Concern> datas) {
    Vector<Vector> view = new Vector<Vector>();
    for (Concern c : datas) {
        Vector v = new Vector();
    }
}
```

```

        v.add(c.getID());
        v.add(c.getPUB_NAME());
        v.add(c.getPUB_LINK_MAN());
        v.add(c.getPUB_TEL());
        v.add(c.getPUB_INTRO());
        view.add(v);
    }
    return view;
}
//实现父类方法，查询数据库并返回对应的数据格式，调用父类的 setDatas 方法设置数据集合
public void setViewDatas() {
    Vector<Concern> concerns = (Vector<Concern>)service.getAll();//使用业务接口得到数据
    Vector<Vector> datas = changeDatas(concerns); //使用业务接口得到数据
    setDatas(datas); //调用父类的 setDatas 方法
}

```

注：在 9.2.7 中，我们将一些代码提到父类 `CommonPanel` 中，因此，在各个界面的构造器中，需要调用 `setJTable` 方法，将子类创建的 `JTable` 设置到父类中。

我们在上面所编写的 `setViewDatas` 方法，同样的也需要在构造器中调用。

`CommonPanel` 构造器：

```

public ConcernPanel(ConcernService service) {
    this.service = service;
    initColumns();//初始化列集合
    setViewDatas();//设置列表数据
    DefaultTableModel model = new DefaultTableModel(null, this.columns); //设置列表
    JTable table = new CommonJTable(model);
    setJTable(table); //调用父类的 setJTable 方法
    //省略其他界面代码
}

```

父类得到数据、`JTable` 对象和列集合后，就可以根据这些对象来创建列表了。创建列表的方法主要是父类的 `initData` 方法，具体请看 9.2.7 章节。效果如图 9.7 所示。

图 9.7 获得全部出版社

### 9.5.4 模糊查询

本例中实现模糊查询功能十分简单，通过出版社的名称进行模糊查询，再将结果集返回到表现层。MySQL 的提供一个 `like` 关键字，可以实现 SQL 的模糊查询。数据访问层与业务层的实现如下。

代码清单：code\book\src\org\crazyit\book\dao\impl\ConcernDaoImpl.java

```
//实现接口 ConcernDao 的方法方法
public Collection<Concern> findByName(String name) {
    //使用 like 关键字
    String sql = "SELECT * FROM T_PUBLISHER pub WHERE pub.PUB_NAME like '%" +
        name + "%' ORDER BY pub.ID DESC";
    //调用父类方法返回结果集
    return getDatas(sql, new Vector(), Concern.class);
}
```

注意以上代码的 `like` 关键字，这是 MySQL 提供的关键字。

代码清单：code\book\src\org\crazyit\book\service\impl\ConcernServiceImpl.java

```
//实现 ConcernService 接口的方法
public Collection<Concern> query(String name) {
    return dao.findByName(name);
}
```

同样地，在业务层中，我们也不需要对结果集进行处理，只需要返回数据即可。

代码清单: code\book\src\org\crazyit\book\ui\ConcernPanel.java

```
//按名字模糊查询
private void query() {
    String name = this.queryName.getText();
    Vector<Concern> concerns = (Vector<Concern>)service.query(name);
    Vector<Vector> datas = changeDatas(concerns); //转换数据格式
    setDatas(datas); //设置数据
    refreshTable(); //刷新列表
}
```

在表现层的 ConcernPanel 中, 通过业务接口得到数据后, 再进行数据格式转换, 调用父类的 setDatas 方法和 refreshTable 方法设置数据和刷新表格。得到的效果如图 9.8 所示。

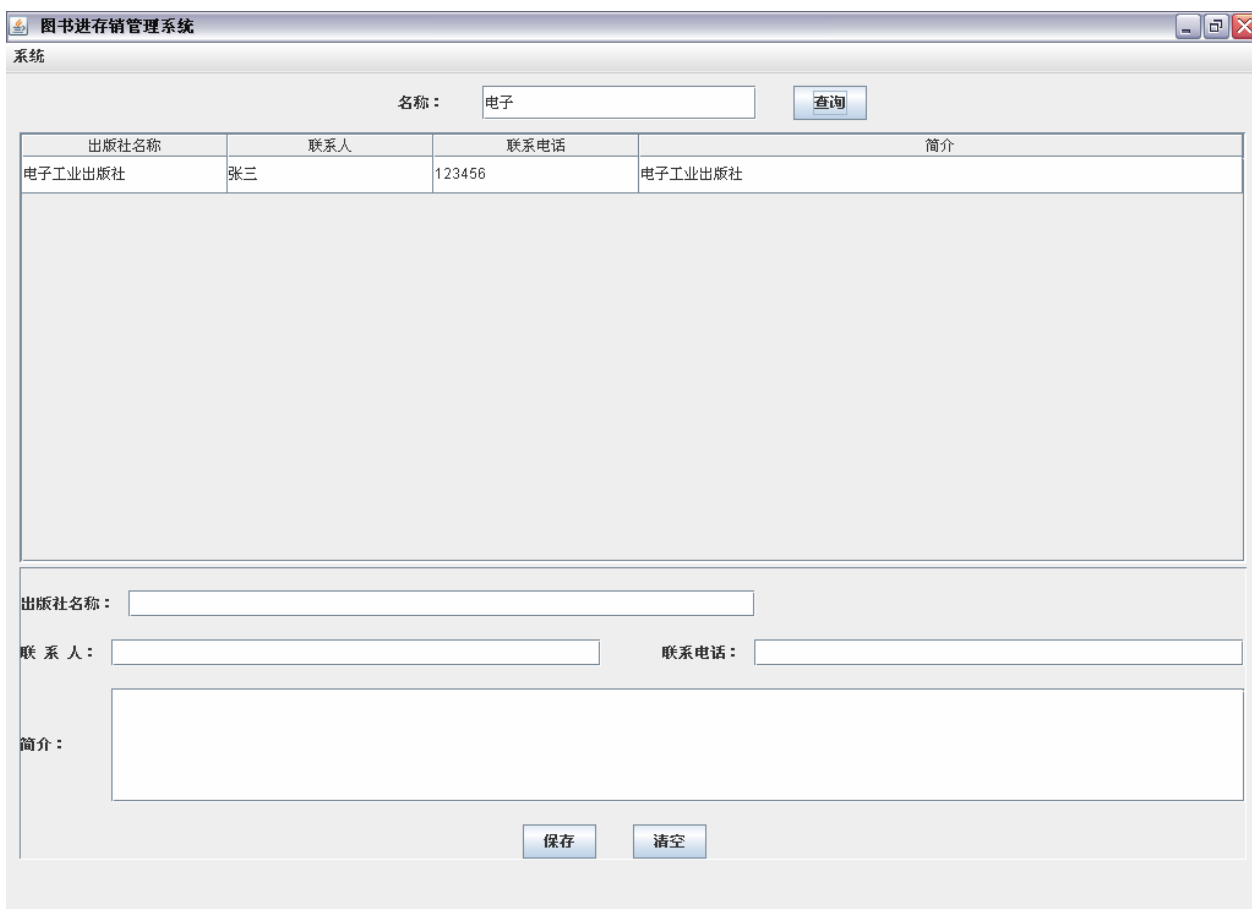


图 9.8 模糊查询

#### 9.5.4 查看出版社

查看出版社, 需要在点击列表的时候, 得到该条数据所对应的 ID, 然后再调用业务层的方法得到数据, 并展现在表单中。当我们在做出版社列表的时候, 需要为列表的 JTable 对象添加一列, 专门用于存放数据的 ID 值, 并设置 ID 列为隐藏, 可以在 setTableFace 方法中设置, 该方法是 CommonPanel 的方法, 需要各个子类去实现。ConcernPanel 的 setTableFace 实现如下。

代码清单: code\book\src\org\crazyit\book\ui\ConcernPanel.java

```
//设置表格样式
public void setTableFace() {
```

```
//隐藏 id 列
getTable().getColumn("id").setMinWidth(-1);
getTable().getColumn("id").setMaxWidth(-1);
getTable().getColumn("简介").setMinWidth(400);
getTable().setRowHeight(30);
}
```

设置了列表的样式后，现在可以为列表的 **Jtable** 添加监听器，当选择一行数据的时候，可以将所选行的信息显示到列表中。**ConcernPanel** 中添加监听器。

代码清单：code\book\src\org\crazyit\book\ui\ConcernPanel.java

```
//查看一个出版社
private void view(String id) {
    //暂时空实现
}
//表格选择监听器
getTable().getSelectionModel().addListSelectionListener(new ListSelectionListener(){
    public void valueChanged(ListSelectionEvent event) {
        //当选择行时鼠标释放时才执行
        if (!event.getValueAdjusting()) {
            //如果没有选中任何一行，则返回
            if (getTable().getSelectedRowCount() != 1) return;
            //调用父类的方法获得选择行的 id
            String id = getSelectId(getTable());;
            view(id);
        }
    }
});
```

注意以上代码的黑体部分，调用了父类（**CommonPanel**）的 **getSelectId** 方法，返回选择行的 **id** 列的值。

当选择了一行数据时，就调用 **view** 方法进行查看，接下来，编写业务层和数据访问层的代码，返回 **Concern** 实体，当表现层得到具体的某个 **Concern** 后，就可以根据它的各个属性，设置到需要显示的地方。

代码清单：code\book\src\org\crazyit\book\dao\impl\ConcernDaoImpl.java

```
//实现 ConcernDao 接口方法
public Concern find(String id) {
    //查询的 SQL
    String sql = "SELECT * FROM T_PUBLISHER pub WHERE pub.ID = " + id + " ";
    //获得数据，并返回第一条
    List<Concern> datas = (List<Concern>)getDatas(sql, new ArrayList(), Concern.class);
    return (datas.size() == 1) ? datas.get(0) : null;
}
```

数据访问层中，与查询所有出版社方法一样，只是改变了一下 **SQL**，得到了相应的数据集，再判断如果数据集的 **size** 不为 1 的话，则返回 **null**。

代码清单：code\book\src\org\crazyit\book\service\impl\ConcernServiceImpl.java

```
//实现接口方法
public Concern find(String id) {
    return dao.find(id);
}
```

业务层与之前的一样，直接返回数据，不必经过任何处理。那么现在就可以去实现表现层的 **view** 方法，得到出版社对象后，再将出版社的各个值设置到对应的地方。

代码清单：code\book\src\org\crazyit\book\ui\ConcernPanel.java

```
//查看一个出版社
private void view(String id) {
    Concern c = service.find(id); //通过业务接口得到出版社对象
    this.concernId.setText(c.getID()); //设置 ID 的 JTextField (隐藏)
    this.concernName.setText(c.getPUB_NAME()); //设置出版社名称的文本框
    this.pubLinkMan.setText(c.getPUB_LINK_MAN()); //设置联系人的文本框
    this.pubTel.setText(c.getPUB_TEL()); //设置联系电话的文本框
    this.pubIntro.setText(c.getPUB_INTRO()); //设置简介的文本域
}
```

查看出版社的功能已经实现，具体效果请见图 9.9。

出版社名称	联系人	联系电话	简介
中国人民出版社	李四	123456	中国人民出版社
电子工业出版社	张三	123456	电子工业出版社

图 9.9 查看出版社

### 9.5.5 新增出版社

接下来实现新增出版社的功能，与查看出版社的功能相反，是从输入框中得到数据，再写进数据库中。但在整个界面中，只有一个保存按钮，也就是说，无论新增还是修改操作，都是使用同一个按钮。什么时候是新增，什么时候是修改，判断的标准是当前编辑的数据是否有 ID，我们在查看出版社的时候，可以看到有一个隐藏的 `JTextField` 来保存出版社的 ID，也就是说，当这个隐藏的 `JTextField` 有值的时候，点击保存，就是修改操作，没有值点击保存按钮，就是新增操作。当用户正在查看一个出版社，又想马上新增一个新的出版社的时候，可以点击清空按钮。

代码清单：code\book\src\org\crazyit\book\ui\ConcernPanel.java



```

//保存方法
private void save() {
    if (this.concernName.getText().trim().equals("")) {
        //调用父类的方法，弹出错误提示
        showWarn("请输入出版社名称");
        return;
    }
    //如果 id 为空，则为新增，不为空则为修改操作
    if (this.concernId.getText().equals("")) add();
    else update();
}
//新增方法
private void add() {
    //暂时不实现
}
//修改方法
private void update() {
    //暂时不实现
}
}

```

为保存按钮添加监听器：

```

//保存按钮监听器
saveButton.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent arg0) {
        save();
    }
});

```

此时，可以去实现业务层和数据访问层。ConcernDao 添加新的接口方法：

```

//添加一个出版社并返回新增的 id
String add(Concern concern);

```

代码清单：code\book\src\org\crazyit\book\dao\impl\ConcernDaoImpl.java

```

public String add(Concern c) {
    //利用出版社对象拼装 SQL
    String sql = "INSERT INTO T_PUBLISHER VALUES (ID, " +
        c.getPUB_NAME() + ", " + c.getPUB_TEL() + ", " + c.getPUB_LINK_MAN() +
        ", " + c.getPUB_INTRO() + ")";
    //调用 JDBCExecutor 的 executeUpdate 方法，并返回新数据的主键
    String id = String.valueOf(getJDBCExecutor().executeUpdate(sql));
    return id;
}

```

以上代码的黑体部分，调用父类的 `getJDBCExecutor()` 方法，获得 `JDBCExecutor` 对象，调用该对象执行 SQL 的 `executeUpdate` 方法，该方法返回新数据的主键。为 `ConcernService` 添加接口方法，新增一个 `Concern` 对象。以下是 `concernServiceImpl` 的实现。

代码清单：code\book\src\org\crazyit\book\service\impl\ConcernServiceImpl.java

```

public Concern add(Concern c) {
    String id = dao.add(c); //得到返回的 ID
    //根据 ID 再查找对象
    return find(id);
}

```

接下来就要表现层将数据封装成一个 `Concern` 对象，再调用业务层的 `add` 方法执行新增操作。

代码清单：code\book\src\org\crazyit\book\ui\ConcernPanel.java

```

//从表单中获取数据并封装成 Concern 对象, 没有 id 值
private Concern getConcern() {
    String concernName = this.concernName.getText();
    String pubTel = this.pubTel.getText();
    String pubLinkMan = this.pubLinkMan.getText();
    String pubIntro = this.pubIntro.getText();
    return new Concern(null, concernName, pubTel, pubLinkMan, pubIntro);
}
//新增方法
private void add() {
    Concern c = getConcern();//获得界面中的值并封装成对象
    service.add(c);
    setViewDatas();//重新读取数据
    clear();//刷新表单
}
//实现父类的方法, 清空表单并刷新列表
public void clear() {
    refreshTable();//调用父类的刷新列表方法
    //清空表单中的各个文本框(域)的值
    this.concernId.setText("");
    this.concernName.setText("");
    this.pubLinkMan.setText("");
    this.pubTel.setText("");
    this.pubIntro.setText("");
}
}

```

注：以上代码中的 `getConcern` 方法，返回时需要用到 `Concern` 的构造器，该构造器有 5 个参数，我们在 `DataUtil` 使用 `Class` 里面的 `newInstance` 方法，因此我们需要为 `Concern` 显示提供一个无参数的构造器。

当调用业务方法新增了一个出版社后，需要将列表刷新，将表单清空，因此我们实现父类的 `clear` 方法，该方法也可以让界面中的清空按钮使用，为清空按钮添加监听器：

代码清单：code\book\src\org\crazyit\book\ui\ConcernPanel.java

```

//清空按钮监听器
clearButton.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent arg0) {
        clear();
    }
});

```

### 9.5.6 修改出版社

在 9.5.5 中，我们已经知道，保存按钮是有两个操作的，至于执行哪一个操作，由存放 ID 的隐藏的 `JTextFiled` 来决定的，并且在 9.5.5 中已经有一个 `update` 方法，只是没有提供实现。修改与新增大致相同，都是读到数据，再执行 SQL 进行修改。为 `ConcernDao` 加入接口方法，用于修改一个 `Concern` 对象。

代码清单：code\book\src\org\crazyit\book\dao\impl\ConcernDaoImpl.java

```

public String update(Concern c) {
    //利用参数拼装修改的 SQL
    String sql = "UPDATE T_PUBLISHER pub SET pub.PUB_NAME=" + c.getPUB_NAME() +
        ", pub.PUB_TEL=" + c.getPUB_TEL() + ", pub.PUB_LINK_MAN=" + c.getPUB_LINK_MAN() +

```

```

        ", pub.PUB_INTRO=" + c.getPUB_INTRO() + " WHERE pub.ID=" + c.getID() + """;
        //执行 SQL
        getJDBCExecutor().executeUpdate(sql);
        //直接返回对象的 ID
        return c.getID();
    }

```

为 ConcernService 接口加入 update 方法，并在 ConcernServiceImpl 中提供实现。

代码清单：code\book\src\org\crazyit\book\service\impl\ConcernServiceImpl.java

```

public Concern update(Concern c) {
    //调用 DAO 方法修改对象
    String id = dao.update(c);
    //重新查找该对象
    return find(id);
}

```

代码清单：code\book\src\org\crazyit\book\ui\ConcernPanel.java

```

//修改方法
private void update() {
    //取得当前修改记录的 ID
    String id = this.concernId.getText();
    //根据界面数据获得 Concern 对象
    Concern c = getConcern();
    c.setID(id);
    service.update(c);
    //重新读取数据
    setViewDatas();
    //刷新列表
    refreshTable();
}

```

注：以上黑体代码为 getConcern 方法，与新增时的 getConcern 方法一样，都是根据界面中各个相关的值获得一个没有 ID 属性的 Concern 对象，update 方法中获得该对象后，由于该对象没有 ID，因此还需要 setID。

出版社的查看、新增和修改功能都已经实现了，由于出版社与其他的表没有任何关联，因此实现起来都比较简单，业务层几乎不需要处理任何业务。和出版社管理相类似的书本种类管理，实现起来比出版社管理更加简单，同样地，书本类型管理也没有任何的外键关联，书本类型表的字段也比出版社表的字段少，因此在本例中省略如何实现书本种类管理。

## 9.6 书本管理功能

与出版社管理功能相比，书本管理复杂一点，因为书与出版社、书本种类相关联，因此在操作书的同时，也需要考虑如何得到这两种数据。另外，书还包括图片，因此对图片的处理上还需要编写一些代码。除了书本记录有外键和需要有图片处理的功能外，其他功能都与出版社管理类似，交互方式也是大体一致的。

### 9.6.1 获取全部书

获得数据库中全部的书，并将其显示在列表，与出版社类似，写一句 SQL 查找得到结果的集合，

再使结果显示在界面中。在 9.2.4 中，书本管理界面对应的是 **BookPanel** 类，同样地，该类也是 **CommonPanel** 的子类，只需要重写一些方法，并设置相关的数据让父类去根据这些数据生成列表即可。首先，我们先实现数据访问层，再实现业务层。新建接口 **BookDao**，并为其提供 **BookDaoImpl** 的实现类，为 **BookDao** 加入 **findAll** 的接口方法，用于查找全部的书本。

代码清单：code\book\src\org\crazyit\book\dao\impl\BookDaoImpl.java

```
public class BookDaoImpl extends CommonDaoImpl implements BookDao {
    public Collection<Book> findAll() {
        //查找的 SQL 语句
        String sql = "SELECT * FROM T_BOOK book ORDER BY book.ID desc";
        //调用父类的方法返回数据
        return getDatas(sql, new Vector(), Book.class);
    }
}
```

与出版社管理类类似，只提供一句 **SQL**，再通过该句 **SQL**，调用父类的 **getDatas** 方法返回数据，**getDatas** 方法对 **ResultSet** 进行了封装。添加一个 **BookService** 的业务接口，并为其提供一个 **BookServiceImpl** 的实现，加入 **getAll** 方法，用于查找全部的书本。该接口给界面对象调用。

在实现 **BookService** 的时候，我们需要知道，**Book** 对象里面包含有书表的各个字段，包括种类（**PUB\_ID\_FK**）和出版社（**TYPE\_ID\_FK**）外键，只是保存一个 **ID**，但是界面上需要显示的是出版社名称和种类名称，因此我们需要为 **Book** 类添加两个属性，分别是种类（**Type**）和出版社（**Concern**）。

代码清单：code\book\src\org\crazyit\book\vo\Book.java

```
//书本种类，从数据库查询出来的时候，这个属性为 null，再通过本类的 TYPE_ID_FK 去设置这个属性
private Type type;
//书对应的出版社，与 type 相同
private Concern concern;
//省略 setter 和 getter 方法
```

因此在 **BookService** 的实现类中，我们需要将两个外键转换成两个对象，这时的转换，则需要 **TypeDao** 和 **ConcernDao** 来根据 **ID** 查找两个对象了。

代码清单：code\book\src\org\crazyit\book\service\impl\BookServiceImpl.java

```
private BookDao bookDao;
private TypeDao typeDao;
private ConcernDao concernDao;
//当构造 BookServiceImpl 的时候，需要三个接口
public BookServiceImpl(BookDao bookDao, TypeDao typeDao, ConcernDao concernDao) {
    this.bookDao = bookDao;
    this.typeDao = typeDao;
    this.concernDao = concernDao;
}
//实现接口方法
public Collection<Book> getAll() {
    Collection<Book> result = bookDao.findAll();
    //调用 setAssociate 方法设置关联的两个对象
    return setAssociate(result);
}
//设置关系对象
private Collection<Book> setAssociate(Collection<Book> result) {
    //遍历结果集合，设置每一个书的对象
    for (Book book : result) {
        //查找出对应的种类，再为书设置种类对象
        book.setType(typeDao.find(book.getType_ID_FK()));
    }
}
```

```

        //查找出对应的出版社，再为书设置出版社对象
        book.setConcern(concernDao.find(book.getPUB_ID_FK()));
    }
    return result;
}

```

以上代码的黑体部分，设置 **Book** 的两个外键对象。这时候，在界面显示需要取得出版社名称和种类名称的话，直接 **book.getType().getTYPE\_NAME()** 即可，下面实现表现层。

代码清单：code\book\src\org\crazyit\book\ui\BookPanel.java

```

//实现父类方法，查询数据库并返回对应的数据格式，调用父类的 setDatas 方法设置数据集
private void setViewDatas() {
    //查找对应的数据
    Vector<Book> books = (Vector<Book>)bookService.getAll();
    //转换显示格式
    Vector<Vector> datas = changeDatas(books);
    //调用父类方法设置结果集合
    setDatas(datas);
}
//将数据转换成视图表格的格式
private Vector<Vector> changeDatas(Vector<Book> datas) {
    Vector<Vector> view = new Vector<Vector>();
    for (Book book : datas) {
        Vector v = new Vector();
        v.add(book.getID());
        v.add(book.getBOOK_NAME());
        v.add(book.getBOOK_INTRO());
        v.add(book.getAUTHOR());
        v.add(book.getType().getTYPE_NAME());
        v.add(book.getConcern().getPUB_NAME());
        v.add(book.getREPERTORY_SIZE());
        v.add(book.getBOOK_PRICE());
        view.add(v);
    }
    return view;
}
}

```

与出版社一样，添加以上两个方法，设置需要显示的列表数据，再设定相关的列即可，以上代码的 **setViewDatas** 需要让构造器去调用。注意，还需要设置一个隐藏的 **id** 列，实现父类的 **setTableFace** 方法即可。具体的实现与出版社管理一样。书本的模糊查询也是使用 **MySQL** 的 **like** 关键字即可实现，得到数据后，再调用父类的 **setDatas** 和 **refreshTable** 方法就可以实现，在本节不再赘述。

### 9.6.2 查看书本

查看书本的功能与查看出版社的功能一样，都是当选择了列表中的某一行数据时，就将数据显示在下面的表单，功能比较简单，只是将各个值放置到对应的区域即可。在这里不详细描述，下面提供根据 **id** 查找书本的 **SQL**：

```
SELECT * FROM T_BOOK book WHERE book.ID=" + id + "";
```

该句 **SQL** 相当的简单，**id** 由界面传入，可以在 **BookPanel** 中调用父类的 **getSelectId** 即可获得，在业务层处理时，当得到一个 **Book** 对象的时候，与获得全部书的功能一样，需要设置 **Book** 对象的两个属性（**Type** 和 **Concern**），重用 9.6.1 中 **BookServiceImpl** 的 **setAssociate** 方法就可以设置，并不需要太复杂的处理。在表现层中，需要提供一个属性，用来保存当前所显示的图片，由于界面中只有一个

显示图片的地方 (JLabel), 因此, 也需要有一个这样的属性来保存图片。

代码清单: code\book\src\org\crazyit\book\ui\BookPanel.java

```
//刷新图片显示的 JLabel
private void refreshImage() {
    this.imageLabel.setIcon(this.currentImage);//imageLabel 为图片显示的 JLabel
}
//查看书本
private void view() {
    String id = getSelectId(getJTable());
    Book book = bookService.get(id);
    //设置各个对应区域的值
    this.bookId.setText(book.getId());
    this.bookName.setText(book.getBOOK_NAME());
    this.price.setText(book.getBOOK_PRICE());
    this.intro.setText(book.getBOOK_INTRO());
    this.author.setText(book.getAUTHOR());
    this.typeComboBox.setSelectedItem(makeType(book.getType()));
    this.concernComboBox.setSelectedItem(makeConcern(book.getConcern()));
    this.currentImage = new ImageIcon(book.getImage_URL());
    this.currentImagePath = book.getImage_URL();
    //重新设置当前显示的图片
    refreshImage();
}
```

注意以上代码的黑体部分, 需要设置种类和出版社下拉框的值。以下是 **makeType** 方法的代码:

```
//创建一个 Type 对象, 用于添加到下拉框中, 该方法中创建的 Type 对象重写了 toString 和 equals 方法
private Type makeType(final Type source) {
    Type type = new Type(){
        public String toString(){
            return source.getType_NAME();
        }
        public boolean equals(Object obj) {
            if (obj instanceof Type) {
                Type t = (Type)obj;
                if (getId().equals(t.getId())) return true;
            }
            return false;
        }
    };
    type.setId(source.getId());
    return type;
}
```

该方法用来创建一个 **Type** 对象, 该对象重写了 **toString** 和 **equals** 方法, 由于在下拉框 (JComboBox) 中, 调用该对象的 **toString** 方法显示, 判断是否选中了某一个下拉框, 调用 **equals** 方法, 因此我们需要重写这两个方法。在 **equals** 方法中, 我们判断两个 **Type** 是否相等的标准是这两个 **Type** 的 ID 是否一致。以下代码初始化下拉框 (显示数据库中所有的 **Type**)。

代码清单: code\book\src\org\crazyit\book\ui\BookPanel.java

```
//从数据库中获取全部的种类并添加到下拉框中
private void addTypes() {
    //调用种类业务接口取得全部的种类
    Collection<Type> types = this.typeService.getAll();
}
```

```

        for (Type type : types) {
            //typeComboBox 是种类下拉框对象
            this.typeComboBox.addItem(makeType(type));
        }
    }
}

```

出版社的下拉框与种类下拉框一样，添加到下拉框里面的每个对象，都需要重写 `toString` 和 `equals` 方法。

### 9.6.3 新增书本

书的表单与出版式社表单不同，需要实现文件上传的功能，其他字段都可以直接使用 SQL 写入到数据库。先将数据访问层和业务层的代码实现。

代码清单：code\book\src\org\crazyit\book\dao\impl\BookDaoImpl.java

```

//实现接口 BookDao 的 add 方法
public String add(Book book) {
    //根据 book 对象拼装 SQL
    String sql = "INSERT INTO T_BOOK VALUES (ID, " + book.getBOOK_NAME() + ", " +
        book.getBOOK_INTRO() + ", " + book.getBOOK_PRICE() + ", " + book.getType_ID_FK() +
        ", " + book.getPUB_ID_FK() + ", " + book.getImage_URL() +
        ", " + book.getAUTHOR() + ", " + book.getREPERTORY_SIZE() + ")";
    //执行 SQL 并返回 ID
    return String.valueOf(getJDBCExecutor().executeUpdate(sql));
}

```

代码清单：code\book\src\org\crazyit\book\service\impl\BookServiceImpl.java

```

//实现 BookService 的 add 方法
public Book add(Book book) {
    String id = bookDao.add(book);
    return get(id);
}

```

代码清单：code\book\src\org\crazyit\book\ui\BookPanel.java

```

//从界面中获取数据并封装成 Book 对象
private Book getBook() {
    String bookName = this.bookName.getText();
    String price = this.price.getText();
    String intro = this.intro.getText();
    String author = this.author.getText();
    //从种类下拉框中取得对应的种类
    Type type = (Type)this.typeComboBox.getSelectedItem();
    //从出版社下拉框中取得相应的出版社
    Concern concern = (Concern)this.concernComboBox.getSelectedItem();
    //this.currentImagePath 是当前显示的图片路径
    return new Book(null, bookName, intro, price, type.getID(),
        concern.getID(), String.valueOf(0), this.currentImagePath, author);
}

//新增书本
private void add() {
    //从界面中取得书本的值并封装成 Book 对象，新增书本时库存为 0
    Book book = getBook();
    bookService.add(book);
}

```



```

//重新读取数据
setViewDatas();
//刷新列表, 清空表单
clear();
}

```

界面上只提供了一个保存的按钮, 点击保存时进行新增还是修改操作, 取决于当前表单中是否有书本, 如果有的话, 即表示修改。在新增书本的时候, 书本的库存应该为 0, 修改书本的时候, 并不应该修改书本的库存, 只有销售或者入库的时候, 才会对库存产生影响。下面详细讲解如何进行图片上传。

#### 9.6.4 图片上传功能

界面提供一个选择图片的按钮, 可以提供给用户选择书的图片, 在本例中, 不必做得太复杂, 因此一本书只有一张图片。在用户选择了图片后, 我们可以将用户所选择的图片复制到我们的项目目录中, 并再生成一张对应的缩略图, 缩略图在书本管理界面显示, 当点击了缩略图时, 可以显示原图给用户浏览。先编写一个 ImageUtil 类, 用来专门处理图片。

代码清单: code\book\src\org\crazyit\book\commons\ImageUtil.java

```

//压缩图片的最大宽
public final static int MAX_WIDTH = 220;
//压缩图片的最大高
public final static int MAX_HEIGHT = 240;
/**
 * 经过原图片的压缩后, 生成一张新的图片
 * @param imageFile 图片文件
 * @param url 原图相对路径
 * @param formatName 生成图片的格式
 * @param compress 是否进行压缩
 * @return 生成的图片文件
 */
public static File makeImage(File imageFile, String url, String formatName,
    boolean compress) {
    File output = new File(url);
    //读取图片
    BufferedImage bi = ImageIO.read(imageFile);
    //如果不进行压缩, 直接写入文件并返回
    if (!compress) {
        ImageIO.write(bi, formatName, new FileOutputStream(output));
        return output;
    }
    //获得新的 Image 对象
    Image newImage = getImage(bi);
    //获取压缩后图片的高和宽
    int height = newImage.getHeight(null);
    int width = newImage.getWidth(null);
    //以新的高和宽构造一个新的缓存图片
    BufferedImage bi2 = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);
    Graphics g = bi2.getGraphics();
    //在新的缓存图片中画图
    g.drawImage(newImage, 0, 0, null);
    //输出到文件

```

```

        ImageIO.write(bi2, formatName, new FileOutputStream(output));
        return output;
    }
    //返回一个 Image 对象, 当参数 bi 的宽比高大的时候, 使用最大的宽来压缩图片。
    //当参数 bi 的高比宽大的时候, 使用最大的高来压缩图片
    private static Image getImage(BufferedImage bi) {
        if (bi.getWidth() > bi.getHeight()) {
            return bi.getScaledInstance(MAX_WIDTH, -10, Image.SCALE_AREA_AVERAGING);
        } else {
            return bi.getScaledInstance(-10, MAX_HEIGHT, Image.SCALE_AREA_AVERAGING);
        }
    }
    //生成 uuid, 作为上传的文件名
    public static String getUUID() {
        UUID uuid = UUID.randomUUID();
        return uuid.toString();
    }
}

```

注意以上代码的黑体部分, 当不需要进行压缩处理的时候, 直接使用 **ImageIO** 输出图像流到新的 url 中, 此处的 url 可以是相对路径。如果需要进行压缩, 那么必须要确定压缩后的图片的高和宽, 因此以上代码中的 **getImage** 方法返回压缩后的 **Image** 对象。下面编写一个文件选择器的类, 处理用户选择文件。

代码清单: code\book\src\org\crazyit\book\ui\BookPanel.java

```

class FileChooser extends JFileChooser {
    //书本管理界面对象
    BookPanel bookPanel;
    public FileChooser(BookPanel bookPanel){
        this.bookPanel = bookPanel;
    }
    //选择了文件后触发
    public void approveSelection() {
        File file = getSelectedFile();//获得选择的文件
        this.bookPanel.upload(file); //调用书本管理界面对象的 upload 方法
        super.approveSelection();
    }
}

```

以上代码的黑体部分, 使用书本管理界面对象的 **upload** 方法, 以下是 **upload** 方法的实现。

代码清单: code\book\src\org\crazyit\book\ui\BookPanel.java

```

//上传图片
public void upload(File selectFile) {
    String uuid = ImageUtil.getUUID();//使用 uuid 生成文件名, 保证文件名唯一
    String smallFilePath = "upload/" + uuid + ".jpg";//缩略图的 url
    String bigFilePath = "upload/" + uuid + "-big.jpg";//原图的 url
    File file = ImageUtil.makeImage(selectFile, smallFilePath, "jpg", true); //生成缩略图
    File source = ImageUtil.makeImage(selectFile, bigFilePath, "jpg", false); //生成原图
    this.currentImage = new ImageIcon(file.getAbsolutePath());//设置界面显示的图片对象
    this.currentImagePath = smallFilePath; //设置界面显示的图片 url
    refreshImage();//刷新图片显示区
}

```

**upload** 方法中, 需要生成两张图片, 一张是原图的副本, 与原图一致, 另外一张是缩略图, 缩略图的高和宽都不大于图片显示区的高和宽。显示可以试下图片上传的功能, 如图 9.10 所示。

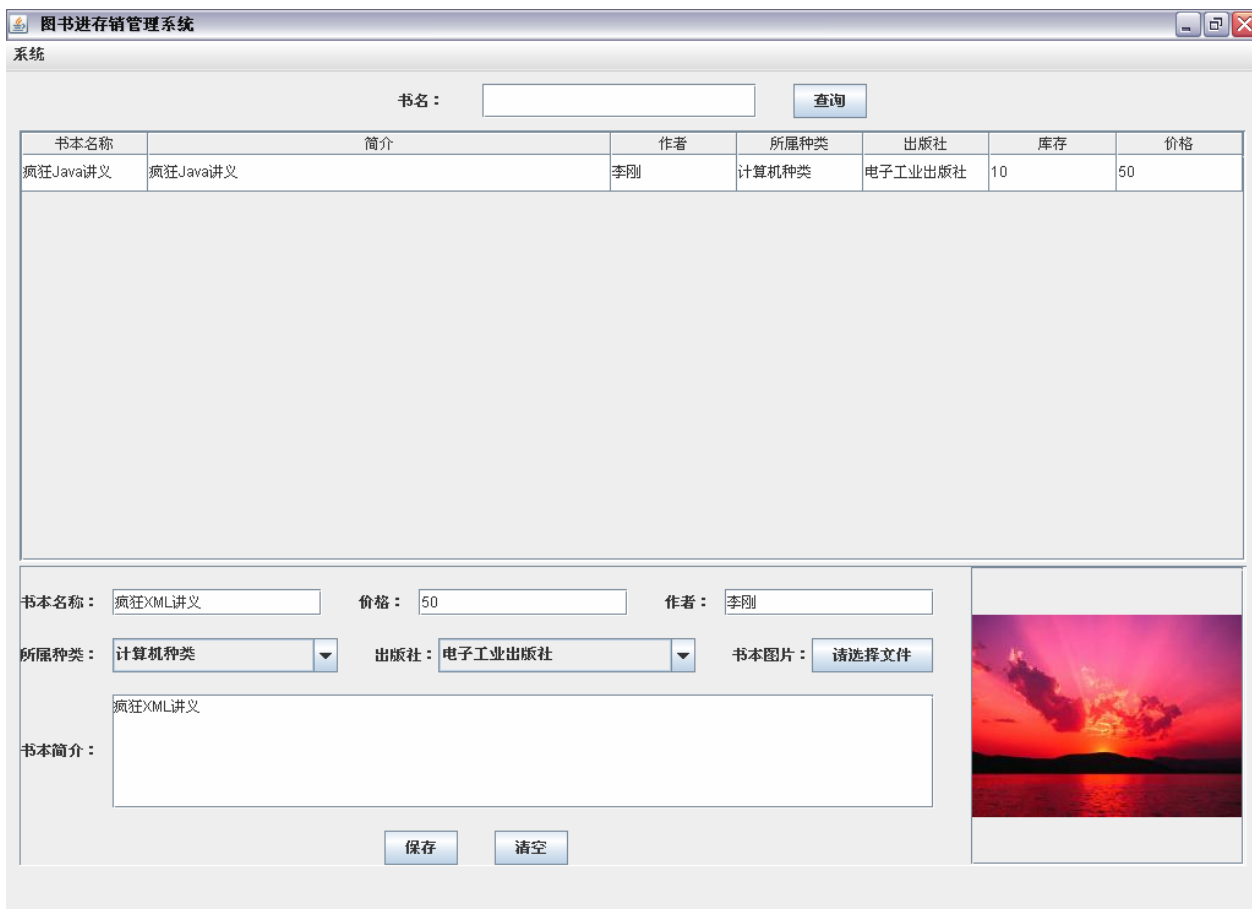


图 9.10 图片上传功能

如果需要浏览原图，可以为图片显示区的 `JLabel` 添加监听器，当点击了该 `JLabel` 时，弹出一个 `JFrame`，显示对应图片的原图。

到此，书本的新增功能已经实现，书本的修改功能与出版社的修改功能一样，图片的上传与显示功能与书本的新增功能一样，在此不再赘述。

## 9.7 销售管理功能

销售管理功能，交互方式与前面几小节的功能类似，但是在表单处理的时候就有所不同，由于一条销售记录可能涉及多本书，因此，在表单中需要有书本交易记录的列表。另外，在为书本交易记录列表增加或者删除一条记录的时候，系统还要为总价和购买数量进行计数量，并显示到相应的地方。当保存销售记录的时候，就需要先保存销售记录，再逐条保存书的销售记录。

### 9.7.1 销售记录列表

查询所有的销售记录，与前面的类似，不同的地方就是查出结果集合后，需要对该集合进行处理，在 9.6 书本管理功能中，查出所有的书本后，还需要根据书本的外键设置书本对象 (`Book`) 的两个关联对象 (`Type` 和 `Concern`)。而在销售管理中，我们还需要为销售记录计算此次交易的总价、交易的书本名称和交易总量。

为 `SaleRecord` 添加属性：

```

//销售的总数量
private int amount;
//总价钱
private double totalPrice;
//书的销售记录
private Vector<BookSaleRecord> bookSaleRecords;
//该记录中对应所有书的名称, 显示用
private String bookNames;
//省略 setter 和 getter 方法

```

为 **SaleRecord** 添加几个属性, 分别是销售总数量、交易金额、书的交易记录和各本书的名称。各本书的名称字符串, 多本书之间使用逗号隔开。剩下的事, 就是在业务层对查出来的 **SaleRecord** 进行处理, 设置我们新增的几个属性。在编写 **SaleRecord** 的业务层时, 需要在书本交易记录的数据访问层中新加一个方法, 根据交易记录的 id 查找到对应的书本交易记录集合。

代码清单: code\book\src\org\crazyit\book\dao\impl\BookSaleRecordDaoImpl.java

```

public class BookSaleRecordDaoImpl extends CommonDaoImpl implements
    BookSaleRecordDao {
    //根据销售记录 id 获得书的销售记录集合
    public Collection<BookSaleRecord> findBySaleRecord(String saleRecordId) {
        String sql = "SELECT * FROM T_BOOK_SALE_RECORD r WHERE r.T_SALE_RECORD_ID_FK=" +
            saleRecordId + "";
        return getDatas(sql, new Vector(), BookSaleRecord.class);
    }
}

```

以上方法, 就可以获得一次交易记录中所有的书的交易记录, 同样的也可以取得交易记录中所涉及的书, 接下来实现销售记录的业务层。

代码清单: code\book\src\org\crazyit\book\service\impl\SaleRecordServiceImpl.java

```

//实现接口方法
public Collection<SaleRecord> getAll(Date date) {
    //得到下一天
    Date nextDate = DateUtil.getNextDate(date);
    //得到今天的日期, 格式为 yyyy-MM-dd
    String today = DateUtil.getDateString(date);
    //得到明天的日期, 格式为 yyyy-MM-dd
    String tomorrow = DateUtil.getDateString(nextDate);
    Collection<SaleRecord> records = saleRecordDao.findByDate(today, tomorrow);
    for (SaleRecord r : records) {
        processDatas(r);
    }
    return records;
}
//处理一个 SaleRecord, 设置它的书本销售记录属性和书本名字属性
private SaleRecord processDatas(SaleRecord r) {
    //查找该记录所对应的书的销售记录
    Collection<BookSaleRecord> brs = bookSaleRecordDao.findBySaleRecord(r.getID());
    //设置结果集中的每一个 book 属性
    setBook(brs);
    //设置 SaleRecord 对象中的书的销售记录集合
    r.setBookSaleRecords((Vector<BookSaleRecord>)brs);
    //设置 SaleRecord 的书名集合
    r.setBookNames(getBookNames(brs));
}

```

```

        //设置数量与总价
        r.setAmount(getAmount(brids));
        r.setTotalPrice(getTotalPrice(brids));
        return r;
    }
    //获取一次交易中涉及的总价
    private double getTotalPrice(Collection<BookSaleRecord> brs) {
        double result = 0;
        for (BookSaleRecord br : brs) {
            //书本的交易量
            int tradeSum = Integer.valueOf(br.getTRADE_SUM());
            //书的单价
            double bookPrice = Double.valueOf(br.getBook().getBOOK_PRICE());
            result += (bookPrice * tradeSum);
        }
        return result;
    }
    //获取一次交易中所有书本的交易量
    private int getAmount(Collection<BookSaleRecord> brs) {
        int result = 0;
        //遍历书的交易记录，计算总价
        for (BookSaleRecord br : brs) {
            result += Integer.valueOf(br.getTRADE_SUM());
        }
        return result;
    }
    //设置参数中的每一个 BookSaleRecord 的 book 属性
    private void setBook(Collection<BookSaleRecord> brs) {
        for (BookSaleRecord br : brs) {
            //调书本的数据访问层接口
            Book book = bookDao.find(br.getBOOK_ID_FK());
            br.setBook(book);
        }
    }
    //获取一次交易中所有书本的名字，以逗号隔开
    private String getBookNames(Collection<BookSaleRecord> brs) {
        if (brids.size() == 0) return "";
        StringBuffer result = new StringBuffer();
        for (BookSaleRecord br : brs) {
            Book book = br.getBook();
            result.append(book.getBOOK_NAME() + ", ");
        }
        //去掉最后的逗号并返回
        return result.substring(0, result.lastIndexOf(", "));
    }
}

```

以上代码中的 `getTotalPrice` 方法用于得到一次交易的价格；`getAmount` 方法用于得到一次交易的书本总数；`setBook` 方法用于设置书本销售记录对象（`BookSaleRecord`）中的 `book` 属性，当然，在那之前要为 `BookSaleRecord` 类添加 `book` 属性；`getBookNames` 方法是将所有交易中涉及的书本的名称集合起来，用逗号隔开。这样，获取全部的交易记录时，得到的每一个交易记录对象（`SaleRecord`），都可以直接 `getBookNames` 或者 `getAmount` 就可以得到交易中涉及书的名称和总数量了。

代码清单: code\book\src\org\crazyit\book\ui\SalePanel.java

```
//将数据转换成主列表的数据格式
private Vector<Vector> changeDatas(Vector<SaleRecord> records) {
    Vector<Vector> view = new Vector<Vector>();
    for (SaleRecord record : records) {
        Vector v = new Vector();
        v.add(record.getID());
        //得到一次交易中所有书的名字
        v.add(record.getBookNames());
        //得到交易的总价
        v.add(record.getTotalPrice());
        v.add(record.getRECORD_DATE());
        //得到交易总数
        v.add(record.getAmount());
        view.add(v);
    }
    return view;
}
```

得到显示的数据后,就可以直接显示在列表中,具体显示请查看 9.5.2 中实现获取全部出版社功能。

### 9.7.2 查看交易记录

查看交易记录,与查看出版社、查看书本等功能类似,只是需要处理书的销售记录的列表显示,即每一次交易记录中所涉及的多本书的记录,都需要在下面的列表显示。先实现数据访问层和业务层,得到 **SaleRecord** 对象,调用 9.7.1 中的 **processDatas** 方法对单个 **SaleRecord** 进行值的处理,分别设置书的名称、交易总价和交易数量,最后返回 **SaleRecord** 对象。以下省略查找 **SaleRecord** 的数据访问层方法和业务层方法,直接对界面进行处理。

代码清单: code\book\src\org\crazyit\book\ui\SalePanel.java

```
//刷新书本销售记录的列表
private void refreshBookSaleRecordTableData() {
    Vector<Vector> view = changeBookSaleRecordDate(this.bookSaleRecordDatas);
    DefaultTableModel tableModel = (DefaultTableModel)this.bookSaleRecordTable.getModel();
    //将数据设入表格 Model 中
    tableModel.setDataVector(view, this.bookSaleRecordColumns);
    //设置表格样式
    setBookSaleRecordTableFace();
}

//查看一条销售记录
private void view() {
    String saleRecordId = getSelectId(getJTable());
    //得到书的交易记录
    SaleRecord record = saleRecordService.get(saleRecordId);
    //设置当前书本销售数据
    this.bookSaleRecordDatas = record.getBookSaleRecords();
    //刷新书本销售列表
    refreshBookSaleRecordTableData();
    this.saleRecordId.setText(record.getID());
    this.totalPrice.setText(String.valueOf(record.getTotalPrice()));
    this.recordDate.setText(record.getRECORD_DATE());
}
```

```

        this.amount.setText(String.valueOf(record.getAmount()));
    }
    //设置书本销售记录的样式
    private void setBookSaleRecordTableFace() {
        this.bookSaleRecordTable.setRowHeight(30);
        //隐藏销售记录 id 列
        this.bookSaleRecordTable.getColumnModel("id").setMinWidth(-1);
        this.bookSaleRecordTable.getColumnModel("id").setMaxWidth(-1);
        //隐藏对应的书 id 列
        this.bookSaleRecordTable.getColumnModel("bookId").setMinWidth(-1);
        this.bookSaleRecordTable.getColumnModel("bookId").setMaxWidth(-1);
    }
    //将书的销售记录转换成列表格式
    private Vector<Vector> changeBookSaleRecordDate(Vector<BookSaleRecord> records) {
        Vector<Vector> view = new Vector<Vector>();
        for (BookSaleRecord r : records) {
            Vector v = new Vector();
            v.add(r.getID());
            v.add(r.getBook().getBOOK_NAME());
            v.add(r.getBook().getBOOK_PRICE());
            v.add(r.getTRADE_SUM());
            v.add(r.getBook().getID());
            view.add(v);
        }
        return view;
    }
}

```

其实销售管理界面比书本管理界面多了一个 **JTable** 对象，用于显示书的销售记录，处理方式与主列表（界面上面的列表）一样，另外，除了需要将书的销售记录放置到下面的 **JTable**，还要为下面的这个 **JTable** 添加两个事件，分别是从此 **JTable** 中添加一本书和删除一本书。以下为两个方法的实现。

代码清单：code\book\src\org\crazyit\book\ui\SalePanel.java

```

//向列表添加一条书的销售记录
private void appendBook() {
    //获得选中的对象
    Book book = (Book)bookComboBox.getSelectedItem();
    String amount = this.bookAmount.getText();
    appendOrUpdate(book, amount);
    //刷新列表
    refreshBookSaleRecordTableData();
    //计算总价
    countTotalPrice();
    //计算总数量
    setTotalAmount();
}
//向书的销售记录列表中删除或者添加一本书的时候计算总价
private void countTotalPrice() {
    double totalPrice = 0;
    for (BookSaleRecord r : this.bookSaleRecordDatas) {
        totalPrice += (Integer.valueOf(r.getTRADE_SUM()) *
            Double.valueOf(r.getBook().getBOOK_PRICE()));
    }
}

```



```
        this.totalPrice.setText(String.valueOf(totalPrice));
    }
    //向书的销售记录列表中删除或者添加一本书的时候设置总数量
    private void setTotalAmount() {
        int amount = 0;
        for (BookSaleRecord r : this.bookSaleRecordDatas) {
            amount += Integer.valueOf(r.getTRADE_SUM());
        }
        this.amount.setText(String.valueOf(amount));
    }
    //添加或者修改书本交易记录中的对象
    private void appendOrUpdate(Book book, String amount) {
        BookSaleRecord r = getBookSaleRecordFromView(book);
        //如果为空, 则为新添加的书, 非空, 则该书已经在列表中
        if (r == null) {
            //创建 BookSaleRecord 对象并添加到数据集中
            BookSaleRecord record = new BookSaleRecord();
            record.setBook(book);
            record.setTRADE_SUM(amount);
            this.bookSaleRecordDatas.add(record);
        } else {
            int newAmount = Integer.valueOf(amount) + Integer.valueOf(r.getTRADE_SUM());
            r.setTRADE_SUM(String.valueOf(newAmount));
        }
    }
    //从列表中移除一条书的销售记录
    private void removeBook() {
        //在集合中删除对应的索引的数据
        this.bookSaleRecordDatas.remove(bookSaleRecordTable.getSelectedRow());
        //刷新列表
        refreshBookSaleRecordTableData();
        //重新计算总价和总数量
        setTotalAmount();
        countTotalPrice();
    }
}
```

以上代码中的 `appendBook` 和 `removeBook` 负责处理添加和删除书本的销售记录列表的数据。以上的 `appendOrUpdate` 方法, 当向列表添加一本书的销售记录时, 如果该本书已经在列表中了, 就去修改该数据, 让书的交易记录的数值加上新输入的数值, 如果该书不在列表中, 则再向里面新增一条记录。得到的效果如图 9.11 所示。

The screenshot displays the '图书进销管理系统' (Library Sales Management System) window. It features a search bar with a date input and a '查询' (Query) button. Below this is a table of transaction records:

购买书本	总价	交易日期	总数量
疯狂XML讲义, 疯狂Java讲义	100.0	2009-10-28 21:33:09.0	2
疯狂Java讲义	100.0	2009-10-28 21:45:00.0	2
疯狂XML讲义	50.0	2009-10-28 21:45:07.0	1

Below the table, there is a summary section with input fields for '总价' (Total Price: 100.0), '交易日期' (Transaction Date: 2009-10-28 21:33:09.0), and '总数量' (Total Quantity: 2). This is followed by another table showing item details:

书名	单价	数量
疯狂XML讲义	50	1
疯狂Java讲义	50	1

At the bottom, there is a form for adding new transactions with fields for '书本' (Book: 疯狂XML讲义), '数量' (Quantity: 1), '单价' (Unit Price: 50), and '库存' (Inventory: 16). It includes '添加' (Add), '删除' (Delete), '成交' (Confirm), and '清空' (Clear) buttons.

图 9.11 实现交易记录的查看功能。

### 9.7.3 实现新增交易记录功能

当在界面中得到各个数据后，即可以调用业务层接口的方法保存交易记录，表现层的数据包括交易日期、各本书的交易信息等。以下为数据访问层（SaleRecordDaoImpl）的实现。

代码清单：code\book\src\org\crazyit\book\dao\impl\SaleRecordDaoImpl.java

```
public String save(SaleRecord r) {
    //执行 SQL 写入数据
    String sql = "INSERT INTO T_SALE_RECORD VALUES(ID, " + r.getRECORD_DATE() + ")";
    return String.valueOf(getJDBCExecutor().executeUpdate(sql));
}
```

由于我们的销售记录表，只是保存了一个交易时间，因此不需要太复杂的处理，但是，由于一次交易涉及多本书，因此我们需要保存书的交易记录，并且是多条。

书的销售记录数据访问层 BookSaleRecordDaoImpl。

代码清单：code\book\src\org\crazyit\book\dao\impl\BookSaleRecordDaoImpl.java

```
public String saveBookSaleRecord(BookSaleRecord record) {
    //保存一条书的销售记录
    String sql = "INSERT INTO T_BOOK_SALE_RECORD VALUES (ID, " + record.getBook().getID() +
        ", " + record.getT_SALE_RECORD_ID_FK() + ", " + record.getTRADE_SUM() + ")";
    return String.valueOf(getJDBCExecutor().executeUpdate(sql));
}
```

保存书本的销售记录，只需要保存对应的销售记录的外键、书本记录外键和交易数量，由于一次交易记录对应多书本的销售记录，因此一条书的销售记录，只需要有一本书的外键。业务层只需要将 **BookSaleRecord**（多个）和 **SaleRecord**（一个）给到两个数据访问层即可。

代码清单：code\book\src\org\crazyit\book\service\impl\SaleRecordServiceImpl.java

```
public void saveRecord(SaleRecord record) {
    //遍历判断书的库存是否不够
    for (BookSaleRecord r : record.getBookSaleRecords()) {
        String bookId = r.getBook().getID();
        Book b = bookDao.find(bookId);
        //当库存不够时,抛出异常
        if (Integer.valueOf(r.getTRADE_SUM()) >
            Integer.valueOf(b.getREPERTORY_SIZE())) {
            throw new BusinessException(b.getBOOK_NAME() + " 的库存不够");
        }
    }
    //先保存交易记录
    String id = saleRecordDao.save(record);
    //再保存书的交易记录
    for (BookSaleRecord r : record.getBookSaleRecords()) {
        //设置销售记录 id
        r.setT_SALE_RECORD_ID_FK(id);
        bookSaleRecordDao.saveBookSaleRecord(r);
        //修改书的库存
        String bookId = r.getBook().getID();
        Book b = bookDao.find(bookId);
        //计算剩余的库存
        int leave = Integer.valueOf(b.getREPERTORY_SIZE()) -
            Integer.valueOf(r.getTRADE_SUM());
        //设置库存并将库存数保存到数据库
        b.setREPERTORY_SIZE(String.valueOf(leave));
        bookDao.updateRepertory(b);
    }
}
```

以上代码保存了销售记录并修改书本的库存量，在进行保存销售记录前，需要进行业务判断，判断书本的库存量是否足够，如果不足够则抛出异常。下面可以实现销售记录的查询，可以根据某一天的时间查询当天所有的销售记录，再进行数据显示，在此不再详细赘述。

到此，销售界面的功能已经完成，与其他功能相比，销售管理界面只是多了一个列表，该列表的处理方法与主列表的处理方式大致相同。我们可以去看一下入库管理界面，与销售管理界面一样，都是通过相同的交互方式得到销售或者入库的目的，但是与之不同的地方是，在处理入库业务的时候，我们同样也需要修改书本的库存，销售是减少库存，入库是增加库存。与销售管理功能一样的是，入库管理界面的下面部分也是有一个入库的书的记录列表，原理与书的交易记录列表一致。入库管理功能的实现，可以参考销售管理功能的实现。入库管理功能如图 9.12 所示。

系统

日期:  查询

入库书本	入库日期	入库数量
疯狂XML讲义, 疯狂Java讲义	2009-10-28 21:33:10.0	20
疯狂Java讲义	2009-10-28 22:40:09.0	5
疯狂XML讲义	2009-10-28 22:40:14.0	2

入库日期: 2009-10-28 22-40-22 总数量: 5

书名	单价	数量
疯狂XML讲义	50	2
疯狂Java讲义	50	3

书本: 疯狂Java讲义 数量: 3 现有: 40 添加 删除

入库 清空

图 9.12 入库管理功能

## 9.8 本章小结

在本章中，我们实现了一个简单的图书进存销系统，学习了使用 JDBC 进行数据库操作，使用 Java 的反射技术封装数据，使用单态模式去获得数据库连接，图片的压缩处理等知识点。在界面中主要使用 JTable 的处理，表现层、业务层与数据访问层的分层结构，大大减低了各层之间的代码耦合。在表现层中简单的使用了模板方法这种设计模式，将各个界面中的共同点提取到父类进行处理。如果需要写更良好的代码，可以对本章中的代码进行重构，并可以应用对应的设计模式。运行本章安全需要将“bin\图书进存销系统\sql”目录下的 BOOK\_SYSTEM.sql 导入你的 MySQL 数据库中，打开“bin\图书进存销系统\book.jar”修改包下的“book.jar\cfg\jdbc.properties”配置文件，最后运行“bin\图书进存销系统\startup.bat”即可，默认的用户名和密码都是 crazyit，本章所使用的 MySQL 版本是 5.0。

## 第 10 章 事务跟踪系统

在企业应用中，我们经常会使用到一些事务跟踪系统，这些事务跟踪系统可以帮助我们有效的对事务进行跟踪、管理。例如有这样一个场景，当某公司的经理接收到一个事务（例如需要处理某张订单或者需要购买某些材料），而这个事务不必由自己亲自去解决的时候，可以使用事务跟踪系统创建一个事务，并将该事务分发到相应的员工去处理，员工接收到这个事务的时候，就可以对事务进行处理，并将每一阶段的处理结果保存到系统中，分发事务的管理者就可以时时刻刻了解事务的进展，对事务起到一定的监控作用。在本章中，我们就使用 Java 开发一个简单的 CS 事务跟踪系统。

### 10.1 确定系统需求

开发事务跟踪系统前，我们需要确定该系统的需求，一旦确定了需求，我们在开发的过程中，就需要按照这些需求进行相关的设计，再按照这些需求去实现相应的功能。如果一开始没有确定好需求，那么在开发的过程中，就会产生一系列的问题，例如做到一定程度的时候，就会有点不知所措，因为业务还没有通。

整个事务跟踪系统的中心是事务，所有的操作都是围绕着事务进行的，管理人创建、分发事务，执行者处理事务，将事务的处理结果反馈给管理人，管理人可以查看事务的进行状况、事务的当前处理人等相关信息，如果事务处理完成，那么管理人就需要在系统外进行一些工作，例如某经理接到一个订单，需要他的手下去进行处理，就可以新建一个事务，分发到他的手下，接收到这个事务的员工，就要为完成这个事务进行工作，执行完该事务后，就将处理结果告诉系统，当经理看到事务的处理结果后，就可以在系统外告诉他的客户，我们已经发货了，请客户查收等。这里需要注意的是，我们并不是要做订单系统，只是做一个简单的事务跟踪系统。

当一个员工接收到一个事务的时候，如果他发现自己最近根本没有时间或者没有能力去处理，那么他可以将事务转发给另外的同事，让其帮忙处理，系统需要记录一条转发记录，让事务管理者可以了解这个情况，可以在这个过程中了解到手下的工作饱和度及处理事情的能力等信息。整个系统的大致需要就有这些，该系统的功能并不复杂，只要确定好需求，开发就更加简单。

### 10.2 建立界面

我们大概了解了系统的需求后，可以根据这些需求创建界面。本章与前面章节中的图书进销管理系统一样，都是使用 Swing 创建系统界面，同样使用 MySQL 作为系统数据库。在本小节，我们先根据需求确定界面与系统交互。

#### 10.2.1 登录界面

如果需要进行事务跟踪系统进行业务操作，需要通过登录进入。用户进行登录，可以让系统知道用户的身份，再根据这个身份去判断哪些功能可以使用或者不可以使用，也就是我们常说的权限问题。权限一般包括功能权限和数据权限，功能权限是指某个用户并不能看到这个功能，从而不能使用这个功能，数据权限是指用户的身份并不能去操作（查看）某些数据。本章的重点并不是权限，因此只需要做简单的权限即可。用户登录界面如图 10.1 所示。



图 10.1 系统登录界面

登录界面十分简单，两个输入框和两个按钮即可，其中需要注意的是，密码框使用的是 JPasswordField 类，并不是 JTextField。

### 10.2.2 我的事务界面

我的事务界面主要显示用户需要处理的事务，并提供一些操作这些事务的按钮，让用户可以对自己的事务进行处理，我的事务功能是每个用户都具有的功能，不需要加任务的权限判断，因此用户一登录事务跟踪系统，首先展现的就是该界面。我的事务界面如图 10.2 所示。



图 10.2 我的事务界面

我的事务界面中，默认显示的是进行中的事务，该界面包括了如下操作：

- ❑ 查询事务：根据事务的不同状态查询相应状态的事务，并显示到列表中。
- ❑ 完成事务：当用户执行完某个事务后，可以点击完成，告诉系统已经处理完该事务了。
- ❑ 转发事务：当用户没有时间或者没有能力执行该事务的时候，可以将该事务转发给其他处理人。
- ❑ 暂时不做：如果用户最近没有时间处理该事务（工作饱和），就可以将事务置为暂时不做状态，

那么管理者就可以知道，该用户没有时间处理。

- ❑ 不做：如果用户觉得这个事务没有存在的意义或者根本就不打算完成这个事务，就可以将该事务的状态设置为“不做”的状态，让管理者知道，用户不会为该事务做任何的工作。

在本系统中，事务包括的状态在下面章节将会详细描述。我的事务界面主要是一个 `JTable` 对象，只需要为该 `JTable` 对象提供 `TableCellRenderer` 的类来渲染列表即可。

### 10.2.3 事务的完成、暂时不做、不做的处理界面

在我的事务界面，提供了完成、暂时不做与不做的任务处理功能，当用户点击了这些功能的时候，就需要提供一个界面，让用户进行相关的说明，例如用户点击了暂时不做，而该用户又想说明为什么暂时不做，因此系统需要提供一个处理界面。事务处理界面如图 10.3 所示。

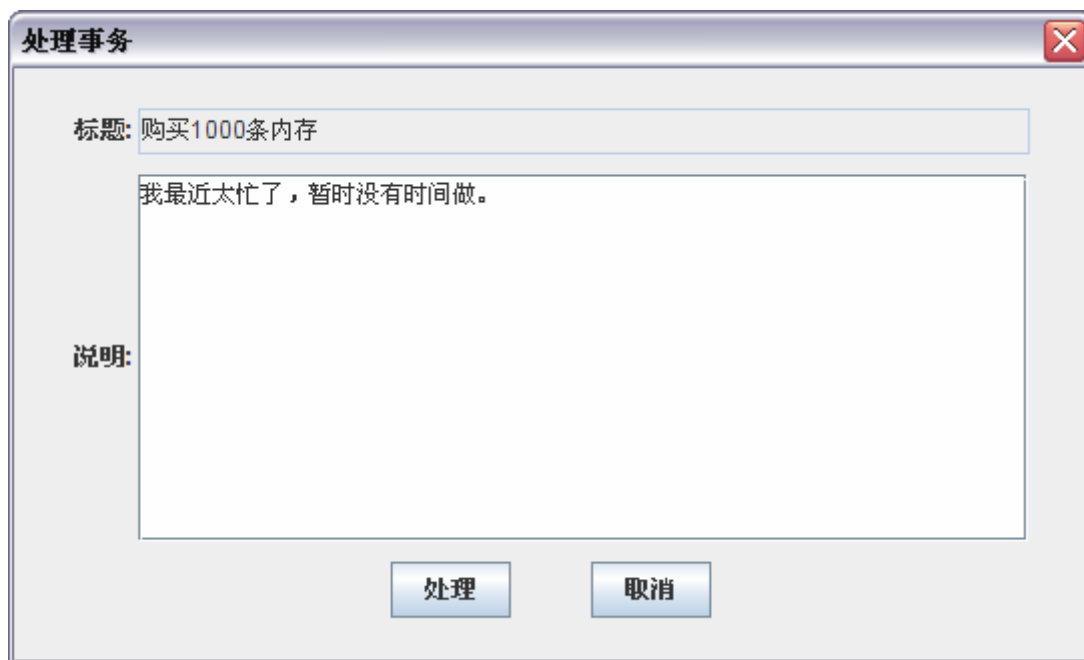


图 10.3 事务处理界面

事务处理界面提供一个 `JTextField` 来显示事务的标题，并提供一个 `JTextArea` 的文本框，让用户描述相关的信息。在这里需要注意的是，用户进行完成、暂时不做与不做的操作时，才会处理事务的界面，在我们的事务界面中，还有一个转发功能，在下面章节中描述。

### 10.2.4 事务转发界面

用户发现自己没有时间或者没有能力去完成该事务的时候，就可以使用事务的转发功能，将属于自己的事务转发给其他同事进行处理。事务转发界面与事务处理界面稍微有一点不同，该界面需要让用户去选择需要转发的人。事务转发界面如图 10.4 所示。



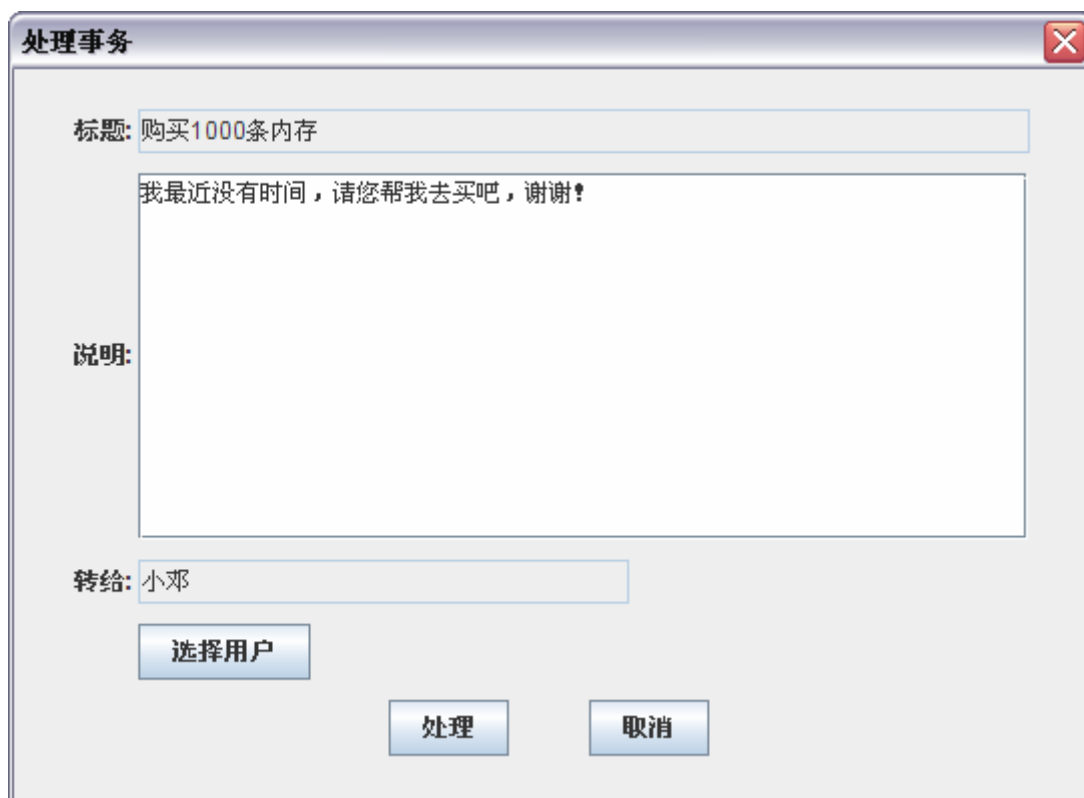


图 10.4 事务转发界面

如图 10.4 所示，在事务转发界面中，有一个选择用户的按钮，当用户点击了选择用户时，就需要弹出用户列表的界面，让用户去选择需要转发的同事，用户选择界面在下面的用户管理章节中将会详细描述。

### 10.2.5 事务管理界面

事务管理界面主要让管理人员进行事务的管理，例如新增事务、查看事务流程等，与我的事务界面一样，提供一个 `JTable` 对象，让管理人员可以清晰的了解各种状态的事务。这里需要注意的是，某一个事务管理者只能看到自己发起的事务，查看到事务的相关状态与流程后，就可以在系统外进行跟进与处理。事务管理界面如图 10.5 所示。

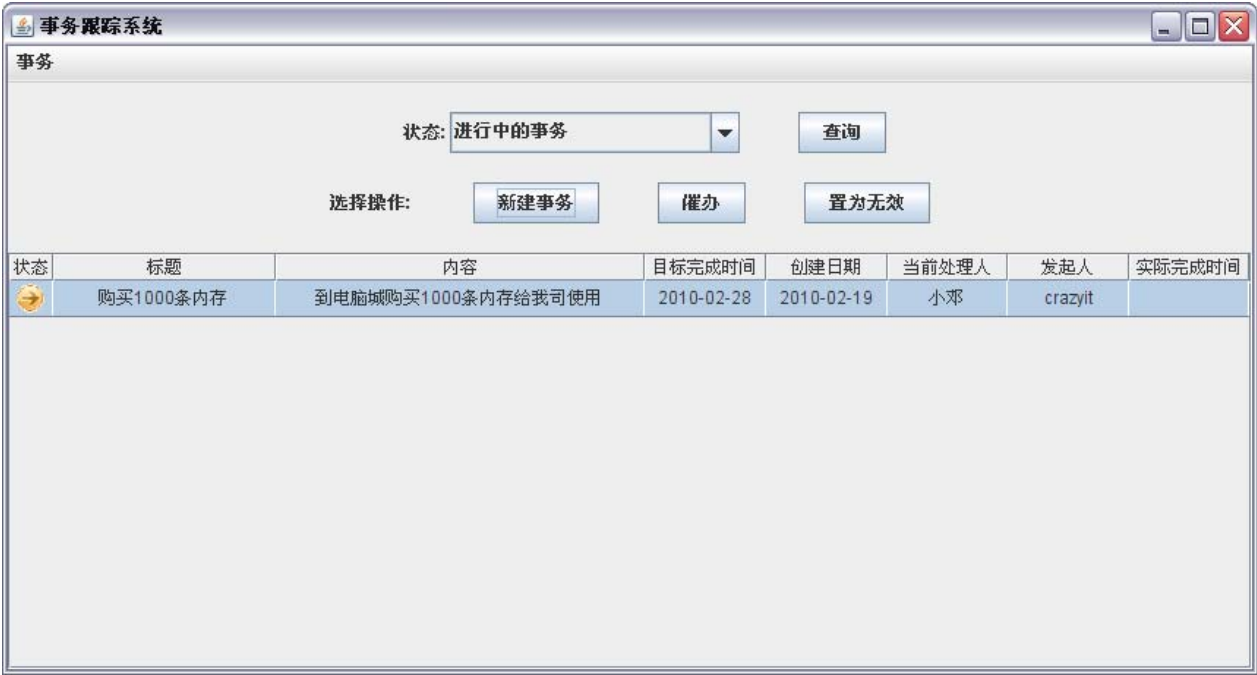


图 10.5 事务管理界面

事务管理界面提供了如下功能：

- ☐ 事务查询：可以查询各种状态的事务。
- ☐ 新建事务：创建一个新的任务，并为其指定处理人。
- ☐ 催办事务：如果一个任务的优先级较高，则可以催办该任务，让任务处理人知道，需要先对该任务进行处理。
- ☐ 置为无效：将一个任务设置为无效，那么任务执行人就可以知道不必为该任务进行任何的工作。

10.2.6 新建事务界面

事务管理者接收到一个新的任务，需要下发到自己的手下进行处理的时候，可以新建一个任务，并指定相关的处理人。新建的任务包括任务的标题、内容、完成时间、处理人等信息，当用户输入了任务的相关信息后，系统就将这些信息保存 to 数据库中。新建任务界面如图 10.6 所示。

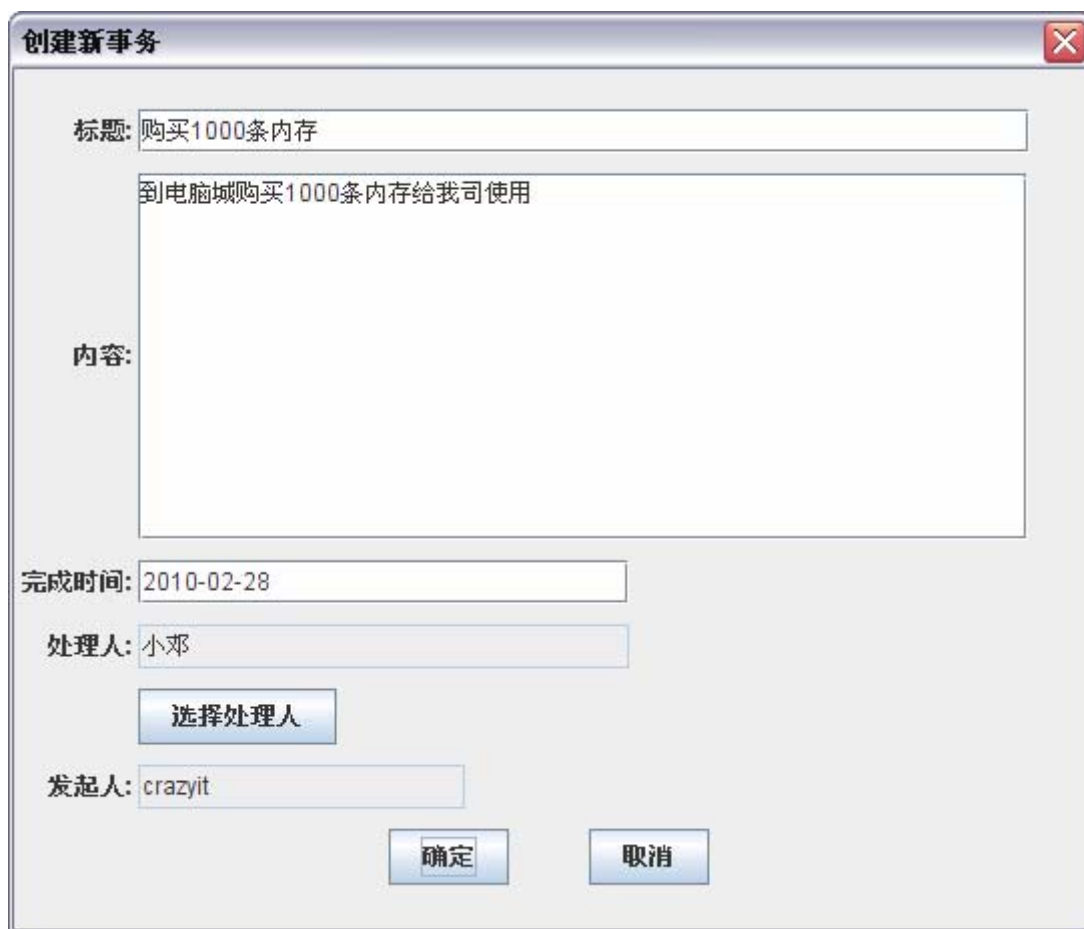


图 10.6 新建事务界面

如图 10.6 所示，新建事务中同样有一个选择用户的按钮，与事务转发界面一样，使用同一个界面作为用户选择界面。

### 10.2.7 用户选择界面

在事务转发界面与新建事务界面中，都有一个用户选择功能，当操作人需要进行用户选择的时候，就需要打开用户选择界面进行选择，在本章中，我们使用同一个界面来实现用户选择的功能。当操作人选择了某个用户后，只需要调用不同的方法即可。新建一个用户选择处理接口。

代码清单：code\transaction\src\org\crazyit\transaction\ui\handler\UserSelectHandler.java:

```
public interface UserSelectHandler {  
    /**  
     * 在用户选择界面点击确定后执行的方法  
     * @param userId  
     * @param realName  
     */  
    void confirm(String userId, String realName);  
}
```

那么在用户选择界面中，只需要提供一个这样的接口即可，操作人选择了某一个用户后，只需要调用接口的 `confirm` 方法即可，不需要理会具体的实现。用户选择界面如图 10.7 所示。



图 10.7 用户选择界面

那么操作人选择了用户后，就可以执行用户选择接口的 `confirm` 方法。

代码清单：`code\transaction\src\org\crazyit\transaction\ui\dialog\SelectUserDialog.java`：

```
//用户选择处理类
private UserSelectHandler selectHandler;
//构造器
public SelectUserDialog(UserSelectHandler selectHandler) {
    this.selectHandler = selectHandler;
    ...
}
//点击确定执行的方法
private void confirm() {
    //得到选择的用户 id 与真实名称
    //调用用户选择处理类的方法
    this.selectHandler.confirm(id, realName);
    this.setVisible(false);
}
```

以上代码是用户选择界面的部分实现，注意以上代码的黑体部分，操作人点击了确定后，就执行 `confirm` 方法。这样，用户选择界面就可以实现重用的功能，因此在本章中，只有一个用户选择界面类。

## 10.2.8 用户管理界面

在实际应用中，还出现一系列的人员变动情况，例如有新的员工入职，有旧的员工离职，因此，我们需要为事务跟踪系统提供一个用户管理界面，让系统管理员可以轻易的管理各个用户。用户管理包括新增用户、删除用户等。用户管理界面如图 10.2.8 所示。



用户名	真实姓名	角色
crazyit	crazyit	admin
zhang	张经理	manager
deng	小邓	employee
deng2	deng2	employee

图 10.2.8 用户管理界面

用户管理界面包括如下功能：

- ❑ 查询用户：根据用户姓名查询相关的用户，在实现中我们使用模糊查询。
- ❑ 新建用户：新建一个用户并分配相应的角色。
- ❑ 删除用户：将该用户从系统中删除，这里需要注意的是，由于原来已经存在的用户可能正在执行某个事务，如果将其从数据库中删除，那么将会影响其他数据，因此我们可以将该用户进行逻辑删除，即不将该用户信息展示到用户管理界面中。

## 10.2.9 新建用户界面

新建一个用户，需要保存用户名（系统用户名）、用户密码、用户真实姓名与分配角色，新建了一个用户后，该用户就拥有了所分配的角色权限，就可以使用本系统中的相关功能。新建用户界面如图 10.2.9 所示。



图 10.9 新建用户

在本章中，为了简单起见，我们内置三种用户角色：管理员、经理与员工，管理员可以使用系统的任何功能，经理不可以使用用户管理功能，员工只能使用我的事务功能。在本小节中，我们制定了系统的各个界面，并确定了系统的交互，界面中的各个组件均使用 **Swing** 的组件，在这里不再详细描述。在下面的章节中，我们将实现这些功能。

## 10.3 开发准备

在本章中，我们将系统分成三层结构，**DAO** 层、业务逻辑层与视图层，**DAO** 层用于进行数据库交互处理，业务逻辑层主要用于处理系统的相关业务，视图层主要是界面组件。如果有使用 **IoC** 容器，我们可以很轻松的对各层之间的组件进行管理，由于本章中并不涉及 **IoC** 容器，因此我们可以使用一个类来对这些组件（**DAO**、业务逻辑）进行管理。

### 10.3.1 数据转换工具类

在开发图书进销管理系统的时候，我们使用 **Java** 的反射来对数据进行封装（详细请看图书进销系统），在本章中，我们同样使用这一方式来对我们的数据进行转换，当执行查询的时候，我们将数据库中读取到的 **ResultSet** 对象转换成具体的某个 **Java** 对象。但有一个前提就是，**Java** 对象中的属性名称必须要与数据库中的表字段名称一致。我们可以将图书进销管理系统的工具拿来使用，以下是图书进销系统的数据转换工具类的具体实现。

代码清单：code\transaction\src\org\crazyit\transaction\util\DataUtil.java:

```
public class DataUtil {
    //将 rs 中的值封装成一个集合
    public static Collection getDatas(Collection result, ResultSet rs, Class clazz) {
        try {
            while (rs.next()) {
                //创建类的实例
                Object vo = clazz.newInstance();
                //获取本对象的属性
                Field[] fields = clazz.getDeclaredFields();
                //获取父类的属性
                Field[] superFields = clazz.getSuperclass().getDeclaredFields();
                //父类的属性和自己的属性相加
            }
        }
    }
}
```

```

        Field[] allFields = addFields(superFields, fields);
        //遍历所有的属性
        for (Field field : allFields) {
            //获得 setter 方法的方法名
            String setterMethodName = getSetterMethodName(field.getName());
            //获得 setter 方法
            Method setterMethod = clazz.getMethod(setterMethodName, field.getType());
            invokeMethod(rs, field, vo, setterMethod);
        }
        result.add(vo);
    }
    rs.close();
} catch (Exception e) {
    e.printStackTrace();
    throw new DataException(e.getMessage());
}
return result;
}
//执行一个方法，从 ResultSet 中获取一个字段的值，调用 vo 的 setter 方法
private static void invokeMethod(ResultSet rs, Field field, Object vo,
    Method setterMethod) {
    try {
        //当使用 ResultSet 获取某个字段的时候，如果没有该字段，会出现 SQLException，忽略该异常
        String value = rs.getString(field.getName());
        if (!"null".equals(value)) {
            //从 ResultSet 中获取与该对象属性名一致的字段，并执行 setter 方法
            setterMethod.invoke(vo, value);
        }
    } catch (Exception e) {
        //忽略异常
    }
}
//根据属性名获得 setter 方法的方法名
private static String getSetterMethodName(String fieldName) {
    String begin = fieldName.substring(0, 1).toUpperCase();
    String end = fieldName.substring(1, fieldName.length());
    String methodName = "set" + begin + end;
    return methodName;
}
//相加两个数组
private static Field[] addFields(Field[] f1, Field[] f2) {
    List<Field> l = new ArrayList<Field>();
    for (Field f : f1) l.add(f);
    for (Field f : f2) l.add(f);
    return l.toArray(new Field[f1.length + f2.length]);
}
}

```

以上代码中的 `getDatas` 方法，遍历参数的 `ResultSet`，将 `ResultSet` 中的某个字段值设置到目标对象中，最后加入到结果集中返回。我们使用了 Java 的反射来找到字段对应的类属性，再调用 `setter` 方法，就可以将字段值设置到具体的对象中。



### 10.3.2 数据库执行类

数据库执行类保存了数据库连接的相关信息，包括连接 IP、用户名、密码等 JDBC 连接信息，提供一些基本的 SQL 执行方法，包括 SQL 的执行、数据库的查询等，并且该类在本系统中只有一个实例。以下是该类的提供的几个基本方法。

代码清单：code\transaction\src\org\crazyit\transaction\jdbc\JDBCExecutor.java:

```
//计算数据总数
public Integer count(String sql) {
    //利用 Statement 对象执行参数的 sql
    ResultSet result = stmt.executeQuery(sql);
    result.next();
    Integer count = result.getInt(1);
    result.close();
    return count;
}
//执行一句查询的 sql
public ResultSet executeQuery(String sql) {
    //利用 Statement 对象执行参数的 sql
    ResultSet result = stmt.executeQuery(sql);
    return result;
}
//执行单句 INSERT、UPDATE 或 DELETE 语句，如果执行 INSERT 时，返回主键
public int executeUpdate(String sql) {
    //执行 SQL 语句
    stmt.executeUpdate(sql);
    //获得主键
    ResultSet rs = stmt.getGeneratedKeys();
    while(rs.next()) {
        //返回最后一个主键
        result = rs.getInt(1);
    }
    rs.close();
    return result;
}
```

以上的代码中使用 JDBC 中的 Statement 执行 SQL 语句得到相应的 ResultSet 对象。

### 10.3.3 DAO层的基类

新建一个 DAO 层，让其他的 DAO 对象都去继续该基类，该基类只提供 getDatas 和 getJDBCExecutor 方法，getDatas 方法对 ResultSet 对象进行封装，getJDBCExecutor 方法返回 JDBC 执行类。

代码清单：code\transaction\src\org\crazyit\transaction\dao\impl\BaseDaoImpl.java:

```
public class BaseDaoImpl {
    //返回 JDBCExecutor 对象
    public JDBCExecutor getJDBCExecutor() {
        return JDBCExecutor.getJDBCExecutor();
    }
    //根据参数的 SQL，存放结果的集合对象，和具体的数据库映射对象返回一个集合
}
```

```
public Collection getDatas(String sql, Collection<ValueObject> result,
    Class clazz) {
    //执行 SQL 返回 ResultSet 对象
    ResultSet rs = getJDBCExecutor().executeQuery(sql);
    //对 ResultSet 进行封装并返回集合
    return DataUtil.getDatas(result, rs, clazz);
}
```

注意以上的黑体代码，使用了数据转换工具类进行数据封装，将 **ResultSet** 对象封装成一个集合返回，提供了该方法后，其他的 DAO 实现类就可以继续这个基类，并使用 **getDatas** 方法对数据进行封装。出可以使用 **getJDBCExecutor** 方法直接得到 JDBC 执行对象，并执行相应的 SQL 语句。

### 10.3.4 值对象基类

在本系统中，我们使用一个值对象来表示一个数据表，并且该值对象中的属性与数据表中的字段名称一致，就好像我们在开发企业应用时所使用的 ORM 框架一样，提供一个 Java 对象与数据库的表进行映射。我们提供一个所有值对象的基类，所有的值对象都需要继续于它。

代码清单：code\transaction\src\org\crazyit\transaction\model\ValueObject.java:

```
public class ValueObject {
    //ID 字段,对应数据库中的 ID 列
    private String ID;
}
```

其他的值对象继承于 **ValueObject**，就意味着该值对象对应的数据库表中必须要有一个 ID 字段。到此，在开发中所使用到的各个类都已经准备完成，接下来可以进行功能开发。

## 10.4 用户管理

从用户管理界面中可以看到，一个用户包括用户名、密码、真实改名与角色，那么在设计数据库的时候，我们可以将这些信息创建一个用户表，用于保存用户记录，另外还需要有一个角色表，用于保存系统中的角色。

### 10.4.1 设计用户管理的相关对象

创建一个角色对象，在系统中表示用户的角色，角色对象只包括角色名称一个属性，由于本章重点并不是权限管理，因此只提供权限名称作为权限标识。

代码清单：code\transaction\src\org\crazyit\transaction\model\Role.java

```
public class Role extends ValueObject {
    //角色名称, 如经理, 总监
    private String ROLE_NAME;
}
```

接下来设计一个用户对象，表示系统中的一个用户对象，用户对象包括用户名、密码、真实姓名与所属的角色外键，还需要注意的是，因为我们要实现删除用户的功能，删除用户时并不是真实的删除用户的数据，而对该用户的数据进行逻辑删除，这是为了不影响其他业务数据而做的，所以还要提供一个标识属性，标识该用户数据是否被删除。

代码清单：code\transaction\src\org\crazyit\transaction\model\User.java:

```
public class User extends ValueObject {
```

```

//用户名称
private String USER_NAME;
//密码
private String PASS_WD;
//用户角色 id, 数据库字段
private String ROLE_ID;
//用户真实名称
private String REAL_NAME;
//是否被删除, 0 没有被删除, 1 为已经删除
private String IS_DELETE;
}

```

设计了用户的两个对象后, 就可以根据这两个对象, 创建相应的数据库表, 在本章中, 如无特别说明, 所有的数据表 ID 均为自动增长 (auto\_increment)。创建角色表与用户表的 SQL 语句如下:

```

CREATE TABLE `t_role` (
  `ROLE_NAME` varchar(255) NOT NULL,
  `ID` int(10) NOT NULL auto_increment,
  PRIMARY KEY (`ID`)
)
CREATE TABLE `t_user` (
  `ID` int(10) NOT NULL auto_increment,
  `USER_NAME` varchar(255) NOT NULL,
  `ROLE_ID` int(10) NOT NULL,
  `REAL_NAME` varchar(255) default NULL,
  `IS_DELETE` varchar(255) NOT NULL default '0',
  `PASS_WD` varchar(255) NOT NULL,
  PRIMARY KEY (`ID`),
  KEY `ROLE_ID` (`ROLE_ID`),
  CONSTRAINT `t_user_ibfk_1` FOREIGN KEY (`ROLE_ID`) REFERENCES `t_role` (`ID`) ON DELETE
  CASCADE
)

```

### 10.4.2 用户登录

新建角色 DAO 与用户 DAO 接口, 并为其提供实现类, DAO 层组件主要负责与数据库进行交互, 编写 SQL 语句让 JDBC 执行类执行, 得到相应的数据后再返回。以下是角色 DAO 接口定义的方法。

代码清单: code\transaction\src\org\crazyit\transaction\dao\RoleDao.java:

```

public interface RoleDao {
    /**
     * 根据 ID 查找角色
     * @param id
     * @return
     */
    Role find(String id);
    /**
     * 查找全部的角色
     * @return
     */
    List<Role> findRoles();
}

```

角色 DAO 接口提供了根据 ID 查找角色与查找全部角色的功能，以下是该接口实现类的具体实现。  
代码清单：code\transaction\src\org\crazyit\transaction\dao\impl\RoleDaoImpl.java:

```
public class RoleDaoImpl extends BaseDaoImpl implements RoleDao {
    public Role find(String id) {
        String sql = "select * from T_ROLE ro where ro.id = " + id + "";
        List<Role> result = (List<Role>)getDatas(sql, new ArrayList(), Role.class);
        return result.size() == 1 ? result.get(0) : null;
    }
    public List<Role> findRoles() {
        String sql = "select * from T_ROLE";
        return (List<Role>)getDatas(sql, new ArrayList(), Role.class);
    }
}
```

在以上代码中，RoleDaoImpl 继承于 DAO 基类：BaseDaoImpl，那么就可以调用基类的两个方法（getDatas 与 getJDBCExecutor），以上代码只负责用于得到相关的角色对象。

新建用户 DAO 接口与其实现类，并新建用户业务逻辑接口 UserService 与其实现类 UserServiceImpl，系统中的相关业务逻辑代码都将被放置到业务逻辑层进行处理，与 DAO 层的实现一样，需要编写一个用户的业务逻辑接口，再为其提供实现。在 DAO 层与业务逻辑层中，我们都需要提供接口，这样做的话，这些组件的调用者，只需要直接使用接口来调用方法，并不需要去关心这些组件是如何实现的，一旦业务或者 DAO 层发生变化，也不必去修改调用者的代码。

业务逻辑层是让视图层调用的，因此我们还需要关心这些业务逻辑对象的创建，我们可以为系统提供一个 ApplicationContext 的对象，来保存这些 DAO 或者业务逻辑组件的实例，那么在系统的任何地方，都可以轻易的得到这些组件的实例。

代码清单：code\transaction\src\org\crazyit\transaction\util\ApplicationContext.java:

```
public class ApplicationContext {
    //登录的用户
    public static User loginUser;
    //用户 DAO 接口
    public static UserDao userDao;
    //角色 DAO 接口
    public static RoleDao roleDao;
    //角色业务逻辑接口
    public static RoleService roleService;
    //用户业务逻辑接口
    public static UserService userService;

    static {
        logDao = new LogDaoImpl();
        roleDao = new RoleDaoImpl();
        roleService = new RoleServiceImpl(roleDao);
        userDao = new UserDaoImpl();
        userService = new UserServiceImpl(userDao, roleDao);
    }
}
```

ApplicationContext 负责创建与维护这些 DAO 组件与业务逻辑组件，使用者只需要使用以下代码，就可以得到某个组件的实例。

ApplicationContext.userService.method;

在 UserDao 中新建一个 findUser 方法，用于根据用户名和密码查找具体的某个用户，并在 UserService 中提供一个 login 的方法，用来进行用户登录，如果用户可以成功登录，则设置到

ApplicationContext 中的 loginUser，表示正在使用系统的用户，如果没有登录成功，则抛出业务异常。

代码清单：code\transaction\src\org\crazyit\transaction\dao\impl\UserDaoImpl.java:

```
public User findUser(String userName, String passwd) {
    String sql = "select * from T_USER u where u.USER_NAME = '" +
        userName + "' and u.PASS_WD = '" + passwd + "' and u.IS_DELETE = '0'";
    List<User> users = (List<User>)getDatas(sql, new ArrayList(), User.class);
    return users.size() == 1 ? users.get(0) : null;
}
```

代码清单：code\transaction\src\org\crazyit\transaction\service\impl\UserServiceImpl.java:

```
public void login(String userName, String passwd) {
    User user = this.userDao.findUser(userName, passwd);
    //没有找到用户，抛出异常
    if (user == null) throw new BusinessException("用户名密码错误");
    Role role = this.roleDao.find(user.getROLE_ID());
    user.setRole(role);
    ApplicationContext.loginUser = user;
}
```

在登录界面中，就可以直接调用 UserService 的 login 方法进行用户登录，并进行异常捕捉。以下是登录界面 LoginFrame 的部分实现。

代码清单：code\transaction\src\org\crazyit\transaction\ui>LoginFrame.java:

```
//点击确定按钮触发的方法
private void login() {
    //得到用户名
    String userName = this.userName.getText();
    //得到密码
    String passwd = getPassword();
    //进行登录
    try {
        ApplicationContext.userService.login(userName, passwd);
        this.setVisible(false);
        MainFrame mf = new MainFrame();
    } catch (Exception e) {
        e.printStackTrace();
        ViewUtil.showWarn(e.getMessage(), this);
    }
}
```

以上的黑体代码中，调用了 UserService 的 login 方法进行登录。

### 10.4.3 查找全部用户

在接口 UserDao 中定义一个 findUsers 方法，用于返回系统中全部的用户。

代码清单：code\transaction\src\org\crazyit\transaction\dao\UserDao.java:

```
/**
 * 查找全部的用户
 * @return
 */
List<User> findUsers();
```

在实现 findUsers 方法的时候，需要注意的是，查询的时候需要加入查询条件，在本章中，用户是可以被逻辑删除的，因此编写 SQL 语句的时候，需要加入逻辑删除的判定条件。

代码清单: code\transaction\src\org\crazyit\transaction\dao\impl\UserDaoImpl.java:

```
public List<User> findUsers() {
    String sql = "select * from T_USER u where u.IS_DELETE = '0'";
    List<User> users = (List<User>)getDatas(sql, new ArrayList(), User.class);
    return users;
}
```

在 `UserService` 中定义 `getUsers` 方法, 用于返回全部的系统用户。以下是 `getUsers` 方法的具体实现。

代码清单: code\transaction\src\org\crazyit\transaction\service\impl\UserServiceImpl.java:

```
public List<User> getUsers() {
    List<User> users = this.userDao.findUsers();
    for (User u : users) {
        Role role = this.roleDao.find(u.getROLE_ID());
        u.setRole(role);
    }
    return users;
}
```

以上代码的黑体部分, 使用 `UserDao` 与 `RoleDao` 来获取数据, 需要注意的是, `User` 对象中有一个 `Role` 属性, 当查出 `User` 对象的时候, 该属性为 `null`, 因此我们需要根据 `User` 的 `ROLE_ID` 属性去获取相应的 `Role` 对象 (到数据库中查找)。在视图层中, 只需要通过 `ApplicationContext` 的 `userService` 就可以调用 `getUsers` 方法得到全部的系统用户。本章的用户列表是一个 `JTable` 的子类, 如果需要对用户列表进行渲染, 只需要提供一个 `TableModel` 即可, 以下是用户列表的 `TableModel` 的部分实现。

代码清单: code\transaction\src\org\crazyit\transaction\ui\table\UserTableModel.java:

```
//表示本列表的数据集合
private List<User> datas;
//返回数据行数
public int getRowCount() {
    if (this.datas != null) {
        return this.datas.size();
    }
    return 0;
}
public String getColumnName(int col) {
    return columnNames[col];
}
public int getColumnCount() {
    return columnNames.length;
}
public void setDatas(List<User> datas) {
    this.datas = datas;
}
//渲染每一列的数据
public Object getValueAt(int row, int column) {
    String columnName = this.getColumnName(column);
    if (this.datas != null) {
        User user = this.datas.get(row);
        if (USER_ID.equals(columnName)) {
            return user.getID();
        } else if (USER_NAME.equals(columnName)) {
```

```

        return user.getUser_NAME();
    } else if (REAL_NAME.equals(columnName)) {
        return user.getREAL_NAME();
    } else if (ROLE.equals(columnName)) {
        return user.getRole().getROLE_NAME();
    }
}
return super.getValueAt(row, column);
}

```

在用户管理界面中只需要调用 `UserService` 的 `getUsers` 方法就可以得到全部的系统用户，再得到 `TableModel` 对象，将用户集合设置到 `TableModel` 中即可。

代码清单：code\transaction\src\org\crazyit\transaction\ui\UserPanel.java:

```

List<User> users = ApplicationContext.userService.getUsers();
this.tableModel.setDatas(users);
this.dataTable.updateUI();

```

这里需要注意的是，无论用户登录还是查找全部用户，都需要加入非逻辑删除的条件判断。

#### 10.4.4 新增用户功能

新增用户，除需要用户名、密码和用户真实姓名外，还需要选择用户的角色。在 `RoleService` 中提供一个 `getRoles` 方法，用于返回全部的角色，在视图层中，只需要调用该方法即可。以下是创建角色下拉框的代码。

代码清单：code\transaction\src\org\crazyit\transaction\ui\dialog\AddUserDialog.java:

```

//创建角色下拉
private void createRoleSelect() {
    this.roleSelect.removeAllItems();
    List<Role> roles = ApplicationContext.roleService.getRoles();
    for (Role r : roles) {
        this.roleSelect.addItem(r);
    }
}

```

以上代码的黑体部分，调用 `RoleService` 的 `getRoles` 方法得到全部的角色，具体的效果如图 10.10 所示。



图 10.10 查找全部角色



在这里需要注意的是, 如果将 **Role** 对象作为下拉框的元素, 需要重写 **Role** 的 **toString** 方法, 返回角色名称。从界面中得到用户的各个信息后, 就可以在 **UserService** 中定义一个 **addUser** 方法, 对新增的用户进行保存, 并在 **UserDao** 中提供一个保存用户的方法, 调用 **JDBC** 执行类来执行插入的 **SQL** 语句。

代码清单: code\transaction\src\org\crazyit\transaction\dao\impl\UserDaoImpl.java:

```
public void save(User user) {
    StringBuffer sql = new StringBuffer("insert into T_USER VALUES (ID, ");
    sql.append(user.getUser_NAME() + ", ");
    .append(user.getROLE_ID() + ", ");
    .append(user.getREAL_NAME() + ", ");
    .append("0, ");
    .append(user.getPASS_WD() + ")");
    getJDBCExecutor().executeUpdate(sql.toString());
}
```

代码清单: code\transaction\src\org\crazyit\transaction\service\impl\UserServiceImpl.java:

```
public void addUser(User user) {
    //根据新的用户名去查找, 判断是否存在相同用户名的用户
    User u = this.userDao.findUser(user.getUser_NAME());
    if (u != null) throw new BusinessException("该用户名已经存在");
    this.userDao.save(user);
}
```

在实现新增用户的业务逻辑时, 如果存在相同用户名的用户数据, 则需要抛出异常, 可以为 **UserDao** 加入一个根据用户查找用户对象的方法, 如果能找到相应的用户, 则表示该用户名已经存在, 不可以新增。

### 10.4.5 删除用户

在本章中, 由于用户有可能与某条业务数据绑定 (事务), 因此不可以随便删除, 如果需要删除数据, 我们只为其提供逻辑删除, 即将用户表的 **IS\_DELETE** 设置为 **1**, 让该数据不会出现在用户管理界面即可。修改数据使用 **SQL** 的 **update** 语句。以下是 **UserDaoImpl** 的实现。

代码清单: code\transaction\src\org\crazyit\transaction\dao\impl\UserDaoImpl.java:

```
public void delete(String id) {
    StringBuffer sql = new StringBuffer("update T_USER u");
    sql.append(" set u.IS_DELETE = '1'");
    .append(" where u.ID = " + id + "");
    getJDBCExecutor().executeUpdate(sql.toString());
}
```

简单的修改用户表的 **IS\_DELETE** 字段值, 再使用 **JDBC** 执行类来执行该句 **SQL** 即可。

### 10.4.6 查询用户

用户管理界面中, 提供了一个查询用户的功能, 操作人输入用户姓名后, 系统就会根据用户所输入的信息到数据库中进行模糊查询。由于我们使用的是 **MySQL** 数据库, 因此可以使用 **MySQL** 的 **like** 关键字进行模糊查询。以下是 **UserDaoImpl** 的查询方法的实现。

代码清单: code\transaction\src\org\crazyit\transaction\dao\impl\UserDaoImpl.java:

```
public List<User> query(String realName) {
    String sql = "select * from T_USER u where u.REAL_NAME like '%"
        + realName + "%' and u.IS_DELETE = '0'";
```

```
List<User> users = (List<User>)getDatas(sql, new ArrayList(), User.class);  
return users;  
}
```

以上代码的黑体部分，使用了 **like** 关键字进行查询，例如要查询用户姓名含有“java”字符串的数据时，就可以使用“**like %java%**”进行查询。

## 10.5 事务管理

事务管理模块是由企业的管理人员进行操作的，在该模块中，可以新建事务并将事务分发到某个下属员工中，还可以查看自己发起的事务状态与流程，如果发现系统中有些事务延后，可以使用模块中的催办功能，让负责该事务的员工看到，该事务需要紧急处理。当管理者发现新建了一些没有意义的事务时，可以将该事务置为无效，这样事务处理人就不必为该事务作出没有必要的处理。

### 10.5.1 设计事务对象

在 10.4.1 中，我们已经设计了角色与用户对象，并根据这些对象创建表结构，那么在设计事务对象的时候，我们也可以采用这样的方式进行设计。从 10.2.6 的新建事务界面中可以看出，一个事务包含的有标题、内容、完成时间、处理人和事务发起人，除了这些信息外，事务还需要保存事务的完成时间、创建时间、事务状态、是否需要紧急处理等信息。创建一个 **Transaction** 的类，继承 **ValueObject**，**Transaction** 对象的属性如下。

代码清单：code\transaction\src\org\crazyit\transaction\model\Transaction.java:

```
public class Transaction extends ValueObject {  
    //事务标题  
    private String TS_TITLE;  
    //事务内容  
    private String TS_CONTENT;  
    //目标完成日期  
    private String TS_TARGETDATE;  
    //实际完成日期  
    private String TS_FACTDATE;  
    //开始日期(创建日期)  
    private String TS_CREATEDATE;  
    //发起人 ID  
    private String INITIATOR_ID;  
    //当前处理人 ID  
    private String HANDLER_ID;  
    //上一个处理人 ID  
    private String PRE_HANDLER_ID;  
    //事务状态  
    private String TS_STATE;  
    //是否需要紧急处理, 0 为不用, 1 需要紧急处理  
    private String IS_HURRY;  
}
```

除了事务的最基本的几个属性外，需要注意的是当前处理人 ID 与上一个处理人 ID，当前处理人是指现在正在执行事务的用户。由于事务可以在用户之间进行转发，因此就存在当前处理人与前一个处理人的问题，为了以后的可扩展性，我们可以建立多一个关联来表示这种关系。以上的 **Transaction** 属性都需要保存到数据库中，根据我们所设计的这些属性，创建事务相应的表。

同样地，也为事务对象创建它的 DAO 对象与业务逻辑对象：TransactionDao、TransactionService。也需要为这两个接口添加实现类。在本小节中，由于只是管理者进行事务管理，并不需要实现其他功能（例如转发），因此不需要设计其他相关的对象。

## 10.5.2 根据发起人查找事务

在事务管理中，事务的管理者可以看到自己发起的事务并对其进行跟踪，当管理者登录到系统中，并进入事务管理时，就需要到数据库中查找该用户所发起的事务。在设计事务对象的时候，就已经提供了一个 INITIATOR\_ID 的属性，表示发起人的 ID，那么我们就可以根据登录用户的 ID 去数据库中进行查找。

为 TransactionDao 加入查询方法，该方法的具体实现如下。

代码清单：code\transaction\src\org\crazyit\transaction\dao\impl\TransactionDaoImpl.java:

```
public List<Transaction> findInitiatorTransactions(String state, String userId) {
    StringBuffer sql = new StringBuffer("select * from T_TRANSACTION ts");
    sql.append(" where ts.INITIATOR_ID = ")
        .append(userId + "")
        .append(" and ts.TS_STATE = ")
        .append(state + " order by ts.IS_HURRY desc");
    List<Transaction> result = (List<Transaction>)getDatas(sql.toString(),
        new ArrayList(), Transaction.class);
    return result;
}
```

由于事务表中存在有发起人的外键，因此可以直接编写 SQL 进行查询，需要注意的是，TransactionDao 提供的查询方法有两个参数，一个是事务的状态，另外一个是用用户的 ID，当用户进行事务查询的时候，该方法也可以重用。以上代码的黑体部分，还需要对查出来的数据进行排序，需要紧急处理的数据将会被放到前面。

从数据库中查询到数据后，就需要将数据展示到界面，事务列表与用户列表的实现一致，都需要提供一个 TableModel。以下是事务列表的部分实现。

代码清单：code\transaction\src\org\crazyit\transaction\ui\table\TransactionTableModel.java:

```
public void setDatas(List<Transaction> datas) {
    this.datas = datas;
}
public List<Transaction> getDatas() {
    return this.datas;
}
public int getRowCount() {
    if (this.datas != null) {
        return this.datas.size();
    }
    return 0;
}
public String getColumnName(int col) {
    return columnNames[col];
}
public int getColumnCount() {
    return columnNames.length;
}
public Object getValueAt(int row, int column) {
    String columnName = this.getColumnName(column);
```

```

        if (this.datas != null) {
            Transaction t = this.datas.get(row);
            if (TS_ID.equals(columnName)) {
                return t.getID();
            } else if (TS_STATE.equals(columnName)) {
                return getStateImage(t.getTS_STATE());
            } else if (TS_TITLE.equals(columnName)) {
                return t.getTS_TITLE();
            } else if (TS_CONTENT.equals(columnName)) {
                return t.getTS_CONTENT();
            } else if (TS_TARGETDATE.equals(columnName)) {
                return t.getTS_TARGETDATE();
            } else if (TS_CREATEDATE.equals(columnName)) {
                return t.getTS_CREATEDATE();
            } else if (CURRENT_HANDLER.equals(columnName)) {
                return t.getHandler().getREAL_NAME();
            } else if (INITIATOR.equals(columnName)) {
                return t.getInitiator().getREAL_NAME();
            } else if (TS_FACTDATE.equals(columnName)) {
                return (t.getTS_FACTDATE() == null) ? "" : t.getTS_FACTDATE();
            }
        }
        return super.getValueAt(row, column);
    }
}

```

在事务列表的 `TableModel` 中维护一个 `List<Transaction>` 的集合，只需要在外界设置这个集合，再调用 `JTable` 的 `updateUI` 方法就可以更新界面的数据。

代码清单：code\transaction\src\org\crazyit\transaction\ui\TransactionManagePanel.java:

```

//从数据库中读取数据
private List<Transaction> getDatas() {
    User loginUser = ApplicationContext.loginUser;
    List<Transaction> datas = ApplicationContext.transactionService
        .getInitiatorTransaction(loginUser, currentState);
    return datas;
}
//事务管理界面读取数据，实现父类的抽象方法
public void readData() {
    List<Transaction> datas = getDatas();
    this.tableModel.setDatas(datas);
    this.dataTable.updateUI();
}

```

事务列表的具体效果如图 10.11 所示。



图 10.11 事务列表

### 10.5.3 新建事务

新建事务意味着一次事务的开始，事务发起人从这时起，就开始对事务进行监控，员工对事务的每一次操作，都需要记录起来，让事务管理者进行查看与跟踪。在 10.2.6 中已经创建了新建事务的界面，并且在 10.5.1 中设计了事务对象，因此我们只需要从界面中得到用户输入的事务信息，再通过 SQL 的 insert 语句写到数据库中。在新建事务的时候，需要注意的是，事务的初始状态是进行中，发起人是进行新建操作的人（不可改变）。

为 TransactionDao 添加方法，以下是保存事务方法的实现。

代码清单：code\transaction\src\org\crazyit\transaction\dao\impl\TransactionDaoImpl.java:

```
public void save(Transaction t) {
    StringBuffer sql = new StringBuffer("insert into T_TRANSACTION values(ID, ");
    sql.append(t.getTS_TITLE() + ", ")
        .append(t.getTS_CONTENT() + ", ")
        .append(t.getTS_TARGETDATE() + ", ")
        .append(t.getTS_FACTDATE() + ", ")
        .append(t.getTS_CREATEDATE() + ", ")
        .append(t.getINITIATOR_ID() + ", ")
        .append(t.getHANDLER_ID() + ", ")
        .append(t.getPRE_HANDLER_ID() + ", ")
        .append(t.getTS_STATE() + ", '0')");
    getJDBCExecutor().executeUpdate(sql.toString());
}
```

以上代码拼装了一句 insert 的 SQL 语句，再通过 JDBC 执行类来执行，需要注意的是新建事务的时候，Transaction 的 IS\_HURRY（是否紧急）的标识为“0”。

在界面中输入了事务的相关信息后，还需要选择事务处理人，在 10.2.7 中创建了用户选择界面，创建了一个 UserSelectHandler 的接口，当新建事务选择处理人时，需要实现这个接口。

代码清单：

code\transaction\src\org\crazyit\transaction\ui\handler\impl\NewTransactionUserSelectHandler.java:

```
public class NewTransactionUserSelectHandler implements UserSelectHandler {
    //新建事务的窗口
    private NewTransactionDialog newDialog;
    public NewTransactionUserSelectHandler(NewTransactionDialog newDialog) {
        this.newDialog = newDialog;
    }
    public void confirm(String userId, String realName) {
        //改变新增窗口的处理人文本框的值
        this.newDialog.getHandlerField().setText(realName);
        this.newDialog.getHandlerIdField().setText(userId);
    }
}
```

以上的 `NewTransactionUserSelectHandler` 类实现了 `UserSelectHandler` 接口，在打开用户选择界面的时候，就使用 `NewTransactionUserSelectHandler` 作为构造参数。

代码清单：code\transaction\src\org\crazyit\transaction\ui\dialog\NewTransactionDialog.java:

```
this.selectHandler = new NewTransactionUserSelectHandler(this);
//创建用户选择对话框
this.userDialog = new SelectUserDialog(this.selectHandler);
```

那么用户选择对话框中使用的其实就是 `NewTransactionUserSelectHandler` 的 `confirm` 方法。`NewTransactionUserSelectHandler` 的 `confirm` 方法只需要改变创建事务界面的两个文本框的值即可。

#### 10.5.4 催办事务

事务管理者发现事务进度落后或者需要紧急处理，就可以使用催办事务的功能，改变事务对象的 `IS_HURRY` 值，在根据发起人查询事务的时候，事务是根据 `IS_HURRY` 属性进行排序的，因此催办事务只需要改变一下这个值即可。

代码清单：code\transaction\src\org\crazyit\transaction\dao\impl\TransactionDaoImpl.java:

```
public void hurry(String id) {
    String sql = "update T_TRANSACTION ts set ts.IS_HURRY = '1' " +
        "where ts.ID = " + id + " ";
    getJDBCExecutor().executeUpdate(sql.toString());
}
```

这样的话，需要紧急处理的事务就会被放到事务列表的最前面，但是仍然不够醒目，因此我们可以在界面层为 `JTable` 添加一个 `TableCellRenderer` 来对列表单元格的样式进行渲染。

代码清单：

code\transaction\src\org\crazyit\transaction\ui\table\TransactionTableCellRenderer.java:

```
public Component getTableCellRendererComponent(JTable table, Object value,
    boolean isSelected, boolean hasFocus, int row, int column) {
    JLabel c = (JLabel)super.getTableCellRendererComponent(table, value,
        isSelected, hasFocus, row, column);
    c.setText("");
    //设置图片
    if (value instanceof ImageIcon) {
        ImageIcon icon = (ImageIcon)value;
        c.setIcon(icon);
        c.setToolTipText(icon.getDescription());
    } else {
```

```

        c.setText(value.toString());
    }
    //设置居中
    c.setHorizontalAlignment(JLabel.CENTER);
    TransactionTableModel model = (TransactionTableModel)table.getModel();
    List<Transaction> datas = model.getDatas();
    if (datas != null) {
        //判断是否需要紧急处理
        if (datas.get(row).getIS_HURRY().equals("1")) {
            c.setFont(hurryFont);
        }
    }
    }
    return c;
}

```

以上代码的黑体部分，得到相应的数据，判断是否需要紧急处理，如果需要的话，就将该行的字体变为粗体，这样可以更加醒目的告诉处理人，这个事务需要紧急处理。具体的效果如图 10.12 所示。



图 10.12 催办事务

如图 10.12 所示，可以看到，排在最前面的事务字体已经为黑体，可以更加醒目的表现该事务需要紧急处理。

### 10.5.5 将事务置为无效

当事务管理者发现事务已经失去其意义，即使已经完成没有任何作用，那么此时可以将事务置为无效，处理人看到该事务已经是无效时，就不必再浪费工作量在该事务上。将事务置为无效实现较为简单，只需要将事务的状态设置到无效的标识即可，但有一点需要注意的是，当事务已经完成时，如果还要将事务置为无效，这显然是没有理由的，因此当用户对已经完成的事务进行置为无效的操作时，系统可以进行判断并拒绝这样的操作。

TransactionDaoImpl:



```

public void invalid(String id) {
    String sql = "update T_TRANSACTION ts set ts.TS_STATE = " +
        TransactionState.INVALID + " where ts.ID = " + id + """;
    getJDBCExecutor().executeUpdate(sql.toString());
}

```

代码清单：code\transaction\src\org\crazyit\transaction\dao\impl\TransactionDaoImpl.java:

```

public void invalid(String id) {
    //如果事务已经完成，则不可以置为无效
    Transaction t = this.transactionDao.find(id);
    if (t.getTS_STATE().equals(TransactionState.FINISHED)) {
        throw new BusinessException("事务已经完成，不可以设置为无效");
    } else {
        this.transactionDao.invalid(id);
    }
}

```

在 `TransactionServiceImpl` 中，先得到一个事务对象，再事务对象的状态进行判断，再决定是否可以进行置为无效处理。

事务管理的功能基本已经实现，但是还有一个查看事务的功能并没有实现，该功能将在下面的章节中详细描述。

## 10.6 我的事务

在 10.5 章节中已经实现了事务管理的相关操作，那么接下来就需要实现员工对事务的处理。在我的事务界面中，同样地有一个事务列表，因此，无论从列表渲染器或者 `TableModel`，都可以使用事务管理的相关对象。

### 10.6.1 设计其他对象

在前面的章节中就已经设计了用户、角色、事务对象，这里所说的其他对象，是指系统所需要使用的其他对象，这些对象包括：事务转发、事务评论与日志对象。用户进行每一次事务转发，都需要将这一次转发的信息保存 to 数据库，还需要保存用户对事务的一些评论，日志是保存员工每一次操作事务的记录，这些日志将会显示到事务查看界面中。

事务转发对象包括以下属性。

代码清单：code\transaction\src\org\crazyit\transaction\model\UserTransfer.java:

```

public class UserTransfer extends ValueObject {
    //事务 ID
    private String TS_ID;
    //进行转发操作的用户
    private String USER_ID;
    //进行转发操作的目标用户
    private String TARGET_USER_ID;
    //进行转发操作用户对该事务的转发时间
    private String OPERATE_DATE;
}

```

事务评论对象包括以下属性。

代码清单：code\transaction\src\org\crazyit\transaction\model\Comment.java:

```

public class Comment extends ValueObject {

```

```

//评论标题
private String CM_TITLE;
//评论内容
private String CM_CONTENT;
//评论日期
private String CM_DATE;
//评论人 ID
private String USER_ID;
//评论的事务
private String TRANSACTION_ID;
}

```

日志对象包括以下属性。

代码清单：code\transaction\src\org\crazyit\transaction\model\Log.java:

```

public class Log extends ValueObject {
    //记录的日期
    private String LOG_DATE;
    //处理人
    private String HANDLER_ID;
    //对应的评论
    private String COMMENT_ID;
    //对应的事务
    private String TS_ID;
    //日志描述
    private String TS_DESC;
}

```

在这里需要注意的是，日志对象中有一个 **COMMENT\_ID**，表示记录一次日志需要有一条评论，在创建界面的时候，我们就已经确定了，对事务进行处理或者转发的时候，都需要给予一定的评论，而這些操作正是我们需要记录日志的地方，因此在设计日志对象的时候，就为日志对象与评论对象设置关联。

查询全部我的事务只需要在 **TransactionDaoImpl** 中提供查询方法即可。

代码清单：code\transaction\src\org\crazyit\transaction\dao\impl\TransactionDaoImpl.java:

```

public List<Transaction> findHandlerTransactions(String state, String userId) {
    StringBuffer sql = new StringBuffer("select * from T_TRANSACTION ts");
    sql.append(" where ts.HANDLER_ID = ")
        .append(userId + "")
        .append(" and ts.TS_STATE = ")
        .append(state + " order by ts.IS_HURRY desc");
    List<Transaction> result = (List<Transaction>)getDatas(sql.toString(),
        new ArrayList(), Transaction.class);
    return result;
}

```

与事务管理中的 **findInitiatorTransactions** 方法不同的是，该方法是根据处理人 ID 去查找事务，而事务管理中是根据发起人 ID 去查找事务。

## 10.6.2 将事务设为“暂时不做”状态

当用户选择了事务并点击了暂时不做的按钮时，系统就需要弹出 10.2.3 中的事务处理界面，让用户输入相应的评论。在本章中我们只有一个事务处理界面，但使用该界面的却有三个功能（完成、暂时不做与不做），因此我们需要像用户选择界面一样，提供一个 **TransactionHandler** 接口，事务处理界面只需要调用该接口的 **handler** 方法即可。

以下是事务处理界面的确定方法。

代码清单：code\transaction\src\org\crazyit\transaction\ui\dialog\HandleTransactionDialog.java:

```
//确定进行事务处理
private void confirm() {
    Comment comment = new Comment();
    comment.setCM_CONTENT(this.content.getText());
    comment.setCM_DATE(ViewUtil.formatDate(new Date()));
    comment.setTRANSACTION_ID(this.transactionId.getText());
    comment.setUSER_ID(ApplicationContext.loginUser.getID());
    try {
        //处理评论与设置事务状态
        this.handler.handler(comment);
        //刷新界面列表
        this.myPanel.readData();
        this.setVisible(false);
    } catch (Exception e) {
        ViewUtil.showWarn(e.getMessage(), this);
    }
}
```

以上代码的黑体部分，只调用 TransactionHandler 的 handler 方法。那么将事务设置为“暂时不做”状态时，就可以使用一个 TransactionHandler 的实现类来构造事务处理界面（HandleTransactionDialog）。

代码清单：code\transaction\src\org\crazyit\transaction\ui\MyTransactionPanel.java:

```
//暂时不做处理类
private TransactionHandler forAWhileHandler = new ForAWhileHandler();
//暂时不做
private void forAWhile() {
    String id = ViewUtil.selectValue(this.dataTable, "id");
    if (id == null) {
        ViewUtil.showWarn("请选择需要操作的事务", this);
        return;
    }
    //得到事务对象
    Transaction t = ApplicationContext.transactionService.get(id);
    //显示处理对话框
    this.htDialog.setTransaction(t);
    this.htDialog.setHandler(this.forAWhileHandler);
    this.htDialog.setVisible(true);
}
```

将事务设置为“暂时不做”的状态，只需要使用 SQL 的 update 语句，改变事务的状态即可。

代码清单：code\transaction\src\org\crazyit\transaction\dao\impl\TransactionDaoImpl.java:

```
public void forAWhile(String id) {
    String sql = "update T_TRANSACTION ts set ts.TS_STATE = " +
        TransactionState.FOR_A_WHILE + " where ts.ID = " + id + " ";
    getJDBCExecutor().executeUpdate(sql.toString());
}
```

在编写业务逻辑的时候，我们需要注意的是，如果用户处理的不是自己的事务，则需要提示，如果事务的状态并不是在进行中，那么就不可以将事务置为“暂时不做”。以下是“暂时不做”的处理类 TransactionHandler 的实现类。ForAWhileHandler:

```

public class ForAWhileHandler implements TransactionHandler {
    public void handler(Comment comment) {
        //将事务置为暂时不做状态，需要先设置事务的状态，再添加评论
        ApplicationContext.transactionService.forAWhile(comment.getTransaction_ID(),
            comment.getUser_ID(), comment);
    }
}

```

以上代码中，调用了事务业务逻辑接口的 `forAWhile` 方法将事务的状态置为“暂时不做”，当用户进行这个操作的时候，就需要将相应的评论保存到数据库，并保存相应的日志信息。

代码清单：code\transaction\src\org\crazyit\transaction\service\impl\TransactionServiceImpl.java:

```

public void forAWhile(String id, String userId, Comment comment) {
    Transaction t = this.transactionDao.find(id);
    //只有自己的事务才可以置为暂时不做状态
    if (!t.getHANDLER_ID().equals(userId)) {
        throw new BusinessException("只能处理自己的事务");
    }
    //只有在进行中的事务才可以改变此状态
    if (t.getTS_STATE().equals(TransactionState.PROCESSING)) {
        this.transactionDao.forAWhile(id);
        //保存评论
        Integer commentId = this.commentDao.save(comment);
        createLog(id, userId, String.valueOf(commentId), "暂时不做");
    } else {
        throw new BusinessException("事务非进行中，不可以置为暂时不做状态");
    }
}
//创建日志
private void createLog(String tsId, String handlerId, String commentId, String desc) {
    Log log = new Log();
    log.setCOMMENT_ID(commentId);
    log.setHANDLER_ID(handlerId);
    log.setLOG_DATE(ViewUtil.timeFormatDate(new Date()));
    log.setTS_ID(tsId);
    log.setTS_DESC(desc);
    this.logDao.save(log);
}

```

以上代码的 `createLog` 方法是工具方法，用于保存一条日志信息，用户进行“暂时不做”的操作时，就会保存一条信息数据与日志数据到数据库。这些数据的作用将在下面的章节中描述。

### 10.6.3 将事务置为“不做”状态

与 10.6.2 中将事务置为“暂时不做”状态类似，只需要改变事务的状态，再保存评论与日志即可。但是事务只能在进行中或者“暂时不做”的状态下才可以将其状态改变为“不做”。事务一旦设置为该状态，将无法恢复到进行中状态，同样地，也不能完成。

为 `TransactionDao` 加入改变状态的方法。

代码清单：code\transaction\src\org\crazyit\transaction\dao\impl\TransactionDaoImpl.java:

```

public void notToDo(String id) {
    String sql = "update T_TRANSACTION ts set ts.TS_STATE = " +
        TransactionState.NOT_TO_DO + " where ts.ID = " + id + "";
}

```

```
getJDBCExecutor().executeUpdate(sql.toString());
}
```

为事务的业务逻辑层加入相关业务方法。

代码清单：code\transaction\src\org\crazyit\transaction\service\impl\TransactionServiceImpl.java:

```
public void notToDo(String id, String userId, Comment comment) {
    Transaction t = this.transactionDao.find(id);
    //只有自己的事务才可以置为暂时不做状态
    if (!t.getHANDLER_ID().equals(userId)) {
        throw new BusinessException("只能处理自己的事务");
    }
    //只有在进行中的事务与暂时不做的事务才可以改变此状态
    if (t.getTS_STATE().equals(TransactionState.PROCESSING)
        || t.getTS_STATE().equals(TransactionState.FOR_A_WHILE)) {
        this.transactionDao.notToDo(id);
        //保存评论
        Integer commentId = this.commentDao.save(comment);
        createLog(id, userId, String.valueOf(commentId), " 决定不做");
    } else {
        throw new BusinessException("不可以置为暂时不做状态");
    }
}
```

同样地，由于使用了同一个事务处理界面，因此就需要提供一个 TransactionHandler 的实现类。

代码清单：code\transaction\src\org\crazyit\transaction\ui\handler\impl\NotToDoHandler.java:

```
public class NotToDoHandler implements TransactionHandler {
    public void handler(Comment comment) {
        //选择将事务状态改变为不做，再添加评论
        ApplicationContext.transactionService.notToDo(comment.getTransaction_ID(),
            comment.getUser_ID(), comment);
    }
}
```

在我的事务界面中，就需要创建处理类。

代码清单：code\transaction\src\org\crazyit\transaction\ui\MyTransactionPanel.java:

```
//决定不做处理类
private TransactionHandler notToDoHandler = new NotToDoHandler();
private void notToDo() {
    String id = ViewUtil.selectValue(this.dataTable, "id");
    if (id == null) {
        ViewUtil.showWarn("请选择需要操作的事务", this);
        return;
    }
    //得到事务对象
    Transaction t = ApplicationContext.transactionService.get(id);
    //显示处理对话框
    this.htDialog.setTransaction(t);
    //设置处理类
    this.htDialog.setHandler(this.notToDoHandler);
    this.htDialog.setVisible(true);
}
```

那么在事务处理界面点击确定的时候，就会执行 NotToDoHandler 的 handler 方法。

### 10.6.4 完成事务

完成事务，只是员工将自己处理的事务置为完成状态，与前面章节中的置为“暂时不做”与“不做”的状态一致，都是需要改变事务状态，再新建评论记录与日志记录。以下是业务逻辑的实现。

代码清单：code\transaction\src\org\crazyit\transaction\service\impl\TransactionServiceImpl.java:

```
//将事务置为完成状态
private void finish() {
    String id = ViewUtil.getSelectedValue(this.dataTable, "id");
    if (id == null) {
        ViewUtil.showWarn("请选择需要操作的事务", this);
        return;
    }
    //得到事务对象
    Transaction t = ApplicationContext.transactionService.get(id);
    //显示处理对话框
    this.htDialog.setTransaction(t);
    this.htDialog.setHandler(this.finishHandler);
    this.htDialog.setVisible(true);
}
```

完成事务提供 TransactionHandler 的实现类 FinishHandler，其他实现均与“暂时不做”与“不做”类似。

### 10.6.5 转发事务

当用户没有时间或者没有能力去处理某个事务的时候，可以将该事务转发给其他的同事，让其代为处理，转发事务的界面与前面的操作稍微有一点不一样，该界面提供了用户选择功能，让操作人选择需要转发的用户，事务转发界面在 10.2.4 中已经创建。在本系统中，只有一个用户选择界面，因此我们按照新建事务时选择用户的方式，提供一个 UserSelectHandler 的实现类 TransferUserSelectHandler，新建事务时的选择用户实现请参看 10.5.3 章节。

代码清单：

code\transaction\src\org\crazyit\transaction\ui\handler\impl\TransferUserSelectHandler.java:

```
public class TransferUserSelectHandler implements UserSelectHandler {
    //转发事务的处理窗口
    private TransferTransactionDialog dialog;
    public TransferUserSelectHandler(TransferTransactionDialog dialog) {
        this.dialog = dialog;
    }
    public void confirm(String userId, String realName) {
        this.dialog.getUserIdText().setText(userId);
        this.dialog.getRealNameText().setText(realName);
    }
}
```

那么在转发事务的界面，需要用户选择处理类。

代码清单：code\transaction\src\org\crazyit\transaction\ui\dialog\TransferTransactionDialog.java:

```
//用户选择处理类
this.selectHandler = new TransferUserSelectHandler(this);
//用户选择对话框
this.userDialog = new SelectUserDialog(this.selectHandler);
```

以上实现了用户的选择,那么当输入完相应转发信息后,点击确定就可以进行保存,以下是业务逻辑的具体实现。

代码清单: code\transaction\src\org\crazyit\transaction\service\impl\TransactionServiceImpl.java:

```
public void transfer(String targetUserId, String sourceUserId,
    Comment comment) {
    Transaction t = this.transactionDao.find(comment.getTRANSACTION_ID());
    //只有自己的事务才可以转发
    if (!t.getHANDLER_ID().equals(sourceUserId)) {
        throw new BusinessException("只能处理自己的事务");
    }
    //只有在进行中的事务与暂时不做的事务才可以转发
    if (t.getTS_STATE().equals(TransactionState.PROCESSING)
        || t.getTS_STATE().equals(TransactionState.FOR_A_WHILE)) {
        UserTransfer ut = new UserTransfer();
        ut.setTS_ID(comment.getTRANSACTION_ID());
        ut.setUSER_ID(sourceUserId);
        ut.setTARGET_USER_ID(targetUserId);
        ut.setOPERATE_DATE(ViewUtil.formatDate(new Date()));
        //新增转发记录
        this.userTransferDao.save(ut);
        //保存评论
        Integer commentId = this.commentDao.save(comment);
        //改变事务记录的当前处理人 id 与前一处理人 id
        this.transactionDao.changeHandler(targetUserId,
            sourceUserId, comment.getTRANSACTION_ID());
        User targetUser = this.userDao.find(targetUserId);
        createLog(t.getId(), sourceUserId, String.valueOf(commentId), " 转 发 给 " +
targetUser.getREAL_NAME() + " ");
    } else {
        throw new BusinessException("只有进行中或者暂时不做的事务才可以转发");
    }
}
```

事务转发并不需要改变事务的状态,只是保存一条转发记录,并将事务的当前处理人设置为新的处理人,将事务对象(Transaction)的前一处理人(PRE\_HANDLER\_ID)设置为原来的处理人 ID。以上代码的黑体部分分别保存转发记录和改变事务的处理人。

### 10.6.6 查看事务

在系统中的每一个用户,都可以查看到自己相关的事务,只需要在事务列表进行双击操作即可,我们并没有在前面的章节中创建查看事务的界面,也没有提供任何查看事务的实现,在本小节,我们将实现查看事务的功能。

当选择了某条事务记录进行查看时,我们需要得到事务的具体信息,除了这些信息外,还需要得到事务的流通过程,例如甲转发了事务给乙,这一过程都需要在查看事务中体现,而员工在操作事务的过程,都会保存一条日志记录,因此进行查看事务的操作时,我们可以将这些日志显示到界面中。以下是查看日志的具体业务实现。

代码清单: code\transaction\src\org\crazyit\transaction\service\impl\TransactionServiceImpl.java:

```
public Transaction view(String id) {
    Transaction t = this.transactionDao.find(id);
    setUser(t);
}
```



```
//查找相应的日志
List<Log> logs = this.logDao.find(id);
for (Log log : logs) {
    Comment comment = this.commentDao.find(log.getCOMMENT_ID());
    User user = this.userDao.find(log.getHANDLER_ID());
    log.setComment(comment);
    log.setHandler(user);
}
t.setLogs(logs);
return t;
}
```

以上代码中，根据事务得到相应的日志，再设置到 **Transaction** 对象中，那么在界面对象中，当得到这个 **Transaction** 后，就可以展现这个对象所保存的相关信息。以下查看事务界面的部分实现。

代码清单：code\transaction\src\org\crazyit\transaction\ui\dialog\ViewTransactionDialog.java:

```
public void setVisible(boolean b) {
    super.setVisible(b);
    if (!b) return;
    //如果当前界面的 Transaction 对象不为空,则设置相应的值
    if (this.transaction != null) {
        this.title.setText(this.transaction.getTS_TITLE());
        this.content.setText(this.transaction.getTS_CONTENT());
        this.targetDate.setText(this.transaction.getTS_TARGETDATE());
        this.handler.setText(this.transaction.getHandler().getREAL_NAME());
        this.initiator.setText(this.transaction.getInitiator().getREAL_NAME());
        this.processArea.setText("");
        for (Log log : this.transaction.getLogs()) {
            this.processArea.append(log.getHandler().getREAL_NAME() + " 于 "
                + log.getLOG_DATE() + " 将事务 " + log.getTS_DESC() + ": "
                + log.getComment().getCM_CONTENT() + "\n");
        }
    }
}
```

以上的黑体代码，将日志组装成相应的字符串，再显示到文本域中，最后的效果如图 10.13 所示。

创建新事务

标题: 购买1000条内存

内容: 请到电脑城购买1000条内存给甲公司使用

完成时间: 2010-03-01

当前处理人: 员工一

发起人: 陈经理

处理过程

员工一 于 2010-02-19 20:25:14 将事务 转发给 员工二 : 你帮我买吧, 我最近没有时间  
员工二 于 2010-02-19 20:25:47 将事务 转发给 员工一 : 我也没有时间, 你自己做吧  
员工一 于 2010-02-19 20:25:59 将事务 暂时不做: 没有时间, 暂时不做  
员工一 于 2010-02-19 20:26:08 将事务 做完了: 已经完成了

关闭

图 10.13 查看事务

可以从图 10.13 中看到，文本域中显示了事务的处理过程，那么事务管理员就可以通过查看事务来了解整个事务的进行状态。

10.6.7 查询事务

事务管理中有一个事务查询功能，该功能根据事务的状态去查询相应的事务。同样地，在我的事务中也有一个事务查询功能，但是该功能与事务管理中的查询功能有不同之处，我的事务的查询功能，状态选择下拉框中，比事务管理中的查询多了一个状态，如图 10.14 所示。

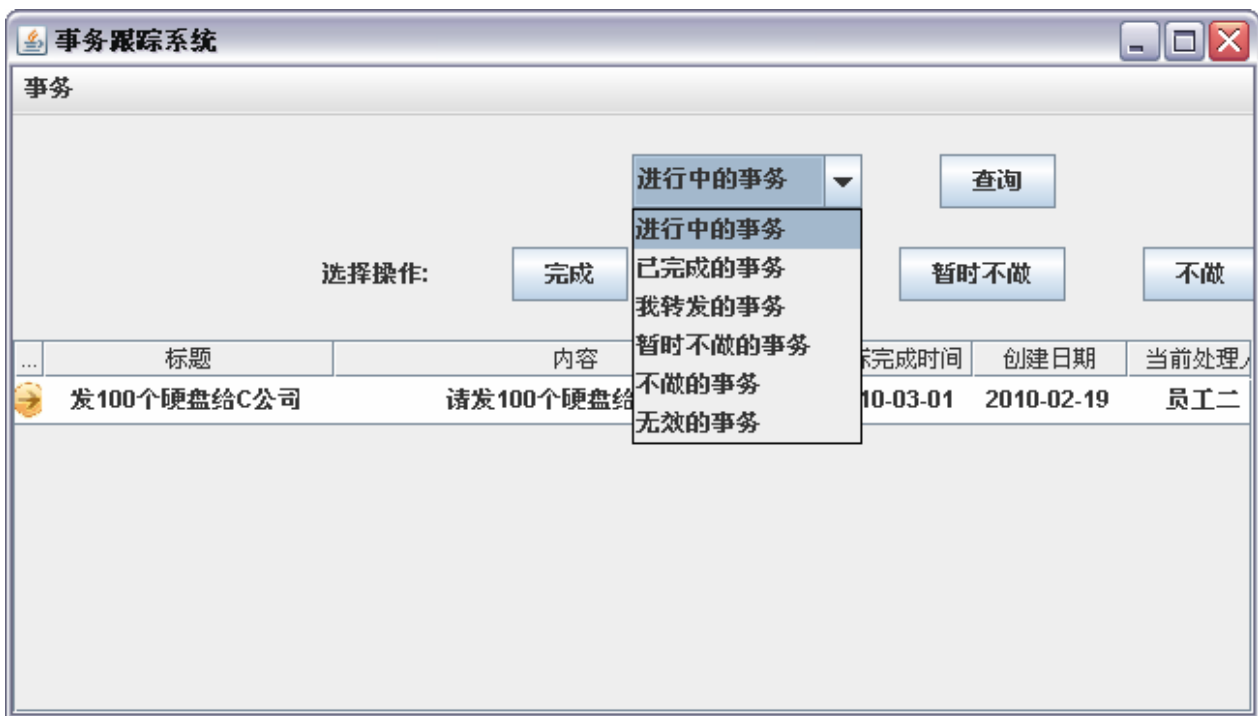


图 10.14 查询事务

我的事务的查询功能下拉框中多出了一个“我转发的任务”这一个状态，当用户选择了该状态进行查询的时候，系统就不是到事务表中查询，而是到转发记录表（UserTransfer）中进行查询，再将查询到的转发记录转换成相应的事务对象。

代码清单：code\transaction\src\org\crazyit\transaction\service\impl\TransactionServiceImpl.java:

```
public List<Transaction> getHandlerTransaction(User user, String state) {
    //如果状态参数是 transfer(转发的任务)，则查找我转发的任务
    if (state.equals(State.TRANSFER)) {
        List<Transaction> datas = new ArrayList<Transaction>();
        //查找转发记录
        List<UserTransfer> transfers = this.userTransferDao.find(user.getID());
        for (UserTransfer ut : transfers) {
            Transaction t = this.transactionDao.find(ut.getTS_ID());
            datas.add(t);
        }
        //去掉重复数据
        datas = removeRepeat(datas);
        return setUnion(datas);
    }
    //其他状态则直接根据状态去数据库查询
    List<Transaction> datas = this.transactionDao.findHandlerTransactions(state,
        user.getID());
    return setUnion(datas);
}
```

以上代码，如果查询的是“我转发的任务”，则先去 UserTransfer 查询所有的转发记录，再通过转发记录得到事务的 ID，就可以找出相应的事务，但是有些情况是一个事务被转发了多次，就产生了多次转发记录，因此还需要通过以上的黑体代码去掉重复的事务数据。

到此，本章的事务跟踪系统的功能已经全部实现。

## 10.7 本章小结

本章开发了一个较为简单的事务跟踪系统，本章所涉及的技术并不难，底层的实现与图书进存销管理系统类似，因此开发的重点放在业务逻辑中，本章描述更多的是事务跟踪系统业务功能的实现。在本章中同样使用了三层结构，当其中一层的逻辑或者实现发生变化时，都不会影响调用者的使用。运行本章的程序，需要修改“bin\事务跟踪系统”中的 `transaction.jar` 包中的配置文件：`transaction.jar\cfg\jdbc.properties`，并将“bin\事务跟踪系统\sql”目录下的 `transaction.sql` 导入你的 MySQL 数据库，最后运行“bin\事务跟踪系统\startup.bat”即可，默认用户名和密码都是 `crazyit`。本章使用的 MySQL 版本是 5.0。

## 第 11 章 多线程下载工具

### 11.1 多线程下载工具简介

我们平时在使用互联网的时候，都会使用到许许多多的下载工具，例如 **Flashget**、迅雷等，这些工具可以十分方便的对互联网资源进行下载。这些工具通过图形界面，将下载的信息展示给用户，这些下载信息包括：下载资源的大小、下载进度、下载速度以及其他一些可以让用户了解下载进度的信息。其实这些下载工具所提供的功能，我们同样可以使用 **Java** 来实现，这些工具包括 **HTTP** 下载、**FTP** 下载以及 **BT** 的支持等，另外，下载工具还需要提供多线程下载、断点续传等功能，本章主要介绍如何实现一个 **HTTP** 的多线程下载工具。

### 11.2 建立下载工具的界面

下载工具包括主界面、添加下载资源的界面、悬浮窗口与系统任务栏图标。下载工具的主界面主要向用户展示下载资源的信息，例如显示正在下载的资源信息、已下载完成的资源信息等。添加下载资源界面主要让用户输入需要下载的资源地址等信息，让下载工具可以对该资源进行连接与下载。悬浮窗口就好像 **Flashget** 与迅雷这些下载工具一样，在屏幕中建立一个小窗口，该窗口可以展现一些下载信息给用户，如图 11.1 所示。系统任务图标主要是将下载工具的图标显示到系统的任务栏中。接下来，我们将介绍如何实现这些相关的界面。



图 11.1 迅雷的悬浮窗口

#### 11.2.1 主界面

下载工具的主界面主要包括导航树、下载列表、资源信息这三大块，导航树我们可以使用 **swing** 的 **JTree** 来实现，下载列表可以使用 **JTable** 实现，资源信息可以使用 **JList** 来实现。

导航树可以让用户选择浏览不同的资源，例如正在下载的资源、下载失败的资源以及成功下载的资源，如果我们需要添加其他的功能，例如对资源进行分类等，都可以向导航树中加入相关的节点。

下载列表主要是将资源的信息展示给用户，包括资源的状态、资源名称、下载速度、进度以及使用时间等。下载列表使用 **JTable** 实现，而下载列表中的每一列都需要进行不同的渲染，例如资源的状态需要显示相应的图片、下载进度需要使用进度条进行显示等。

资源信息一块使用 **JList** 实现，主要将资源的部分信息显示到该区域。本章中主界面对应的是 **MainFrame.java**，主界面的最终效果如图 11.2 所示。

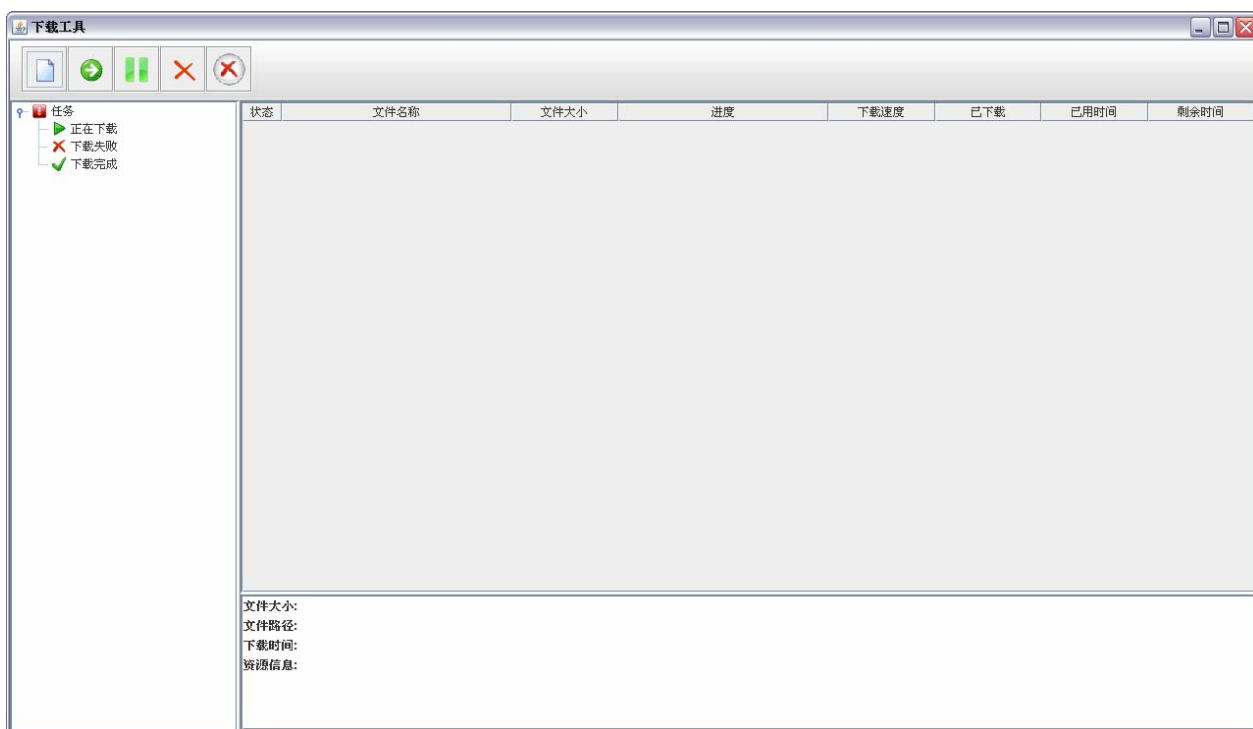


图 11.2 主界面效果

如图 11.2 所示，主界面除了上述的三大区域外，还提供了一个工具栏，用于存放对资源的相关操作：新建资源、开始下载、暂停下载、删除选中资源和删除已经完成的任務。为了达到我们所需要的效果，主界面三个区域（导航树、资源列表和资源信息）都需要进行特别的处理，以下的章节将详细描述如何实现这三个区域。

### 11.2.2 导航树

如图 11.2 所示，导航树主要有一个任务节点，任务节点下面包括正在下载节点、下载失败节点和下载完成节点。在建立导航树前，我们需要准备表示这些节点的对象。新建一个 `DownloadNode` 接口，用于表示导航树的节点，该树下面所有的节点对象都必须实现这个接口。

代码清单：code\flashget\src\org\crazyit\flashget\navigation\DownloadNode.java:

```
public interface DownloadNode {  
    /**  
     * 获得节点名称  
     * @return  
     */  
    String getText();  
    /**  
     * 返回对应图标  
     * @return  
     */  
    ImageIcon getIcon();  
}
```

该接口定义了两个方法，分别返回节点的文字与图标，那么可以新建一个正在下载的节点对象，实现 `DownloadNode`。

代码清单：code\flashget\src\org\crazyit\flashget\navigation\DownloadingNode.java:

```
public class DownloadingNode implements DownloadNode {
    public ImageIcon getIcon() {
        return ImageUtil.DOWNLOADING_NODE_IMAGE;
    }
    public String getText() {
        return "正在下载";
    }
}
```

**DownloadingNode** 表示一个正在下载节点，以同样的方法，新增其他节点对应的实现类，那么在创建导航树的时候，就可以为导航树设置渲染器。在树的渲染器中，我们可以得到相应的节点对象（**DownloadNode** 的实现类），再根据这些对象得到文本与图标。

代码清单：code\flashget\src\org\crazyit\flashget\ui\NavigationTreeCellRender.java:

```
public Component getTreeCellRendererComponent(JTree tree, Object value,
        boolean sel, boolean expanded, boolean leaf, int row,
        boolean hasFocus) {
    super.getTreeCellRendererComponent(tree, value, sel,
        expanded, leaf, row, hasFocus);
    //得到树节点
    DefaultMutableTreeNode node = (DefaultMutableTreeNode)value;
    //得到节点对象
    DownloadNode obj = (DownloadNode)node.getUserObject();
    //设置文本与图片
    if (obj != null) {
        this.setIcon(obj.getIcon());
        this.setText(obj.getText());
    }
    return this;
}
```

在创建导航树的时候，需要树指定渲染器，并创建各个节点。

代码清单：code\flashget\src\org\crazyit\flashget\ui\MainFrame.java:

```
//主界面中创建树
private void createTree() {
    DefaultMutableTreeNode root = new DefaultMutableTreeNode();
    DefaultMutableTreeNode tn = new DefaultMutableTreeNode(taskNode);
    root.add(tn);
    //创建各个节点
    tn.add(new DefaultMutableTreeNode(downloadingNode));
    tn.add(new DefaultMutableTreeNode(failNode));
    tn.add(new DefaultMutableTreeNode(finishNode));
    this.navTree = new NavigationTree(root);
}
```

以上代码的黑体部分，我们使用了一个 **NavigationTree** 的对象来表示一棵导航树，该类继承于 **JTree**，主要用于设置导航树的相关属性，例如隐藏根节点、设置渲染器等。

注意：表面上任务节点是根节点（如图 11.2 所示），但实际上，任务节点只是根节点的一个子节点。

### 11.2.3 资源列表

在资源列表中，每一列都需要有特定的渲染器，例如状态列需要使用图标作为显示内容，下载进度列需要使用进度条作为显示内容。在本章中，我们使用一个“进度条的子类”作为下载进度列的渲染器，

其他的列则使用同一个渲染器。以下是进度条渲染器的实现。

代码清单：code\flashget\src\org\crazyit\flashget\ui\DownloadProgressBar.java:

```
public class DownloadProgressBar extends JProgressBar implements
    TableCellRenderer {
    public DownloadProgressBar() {
        super(0, 100);
        this.setStringPainted(true);
        this.setForeground(Color.green);
    }
    public Component getTableCellRendererComponent(JTable table, Object value,
        boolean isSelected, boolean hasFocus, int row, int column) {
        Float floatValue = Float.parseFloat(value.toString());
        int intValue = (int)floatValue.floatValue();
        this.setValue(intValue);
        this.setString(value.toString() + " %");
        return this;
    }
}
```

以上的 `DownloadProgressBar` 类，继承于 `JProgressBar` 并实现 `TableCellRenderer`，表示这个类是一个进度条，同是也是一个列表的渲染器。下面新建其他列的渲染器，该渲染器负责列表中其他列的渲染。

代码清单：code\flashget\src\org\crazyit\flashget\ui\DownloadTableCellRenderer.java:

```
public Component getTableCellRendererComponent(JTable table, Object value,
    boolean isSelected, boolean hasFocus, int row, int column) {
    //判断是否需要显示图片
    if (value instanceof Icon) this.setIcon((Icon)value);
    else this.setText(value.toString());
    //判断是否选中
    if (isSelected) super.setBackground(table.getSelectionBackground());
    else setBackground(table.getBackground());
    //设置居中
    this.setHorizontalAlignment(JLabel.CENTER);
    this.setToolTipText(value.toString());
    return this;
}
```

创建了两个渲染器后，我们可以在创建列表的时候，将对应的列设置相应的渲染器。在本章，下载列表对应的是 `DownloadTable` 类，该类继承于 `JTable`，`DownloadTable` 类主要负责设置列表的样式。

在 `DownloadTable` 中设置单元格渲染器。

代码清单：code\flashget\src\org\crazyit\flashget\ui\DownloadTable.java

```
//设置单元格渲染
this.getColumnModel().getColumn(DownloadTableModel.STATE_COLUMN).setCellRenderer(
    new DownloadTableCellRenderer());
this.getColumnModel().getColumn(DownloadTableModel.PLAN_COLUMN).setCellRenderer(
    new DownloadProgressBar());
...省略其他列
```

除了进度条列之外，其他的列都是使用 `DownloadTableCellRenderer` 作为列的渲染器。在本章中，`DownloadTable` 并不负责处理数据。在下载的过程中，许多的数据都是在不停的变化，因此我们可以提供一个 `TableModel` 来负责处理列表中的数据。本章所提供的 `TableModel`，包括了声明下载列表的列信息，保存列表数据等。以下是该 `TableModel` 类的部分实现。



代码清单: code\flashget\src\org\crazyit\flashget\ui\DownloadTableModel.java:

```
public int getRowCount() {
    return 0;
}

public String getColumnName(int col) {
    return columnNames[col];
}

public int getColumnCount() {
    return columnNames.length;
}

public Object getValueAt(int row, int column) {
    String columnName = this.getColumnName(column);
    //对列名进行判断, 再返回相应的数据
    return super.getValueAt(row, column);
}
```

DownloadTableModel 继承于 DefaultTableModel 类, 以上代码的 columnNames 是列名数组, 我们需要重写 DefaultTableModel 的 getRowCount、getColumnName、getColumnCount 和 getValueAt 方法, 其中 getRowCount 方法返回数据记录数, getValueAt 方法返回每一列相应的数据。例如资源状态列, 我们需要返回一个图标对象, 进度条列需要返回一个 float 值。到此, 数据列表的各个对象都已经创建, 我们可以在 DownloadTableModel 是新建一些测试数据来查看具体的效果。效果如图 11.3 所示。

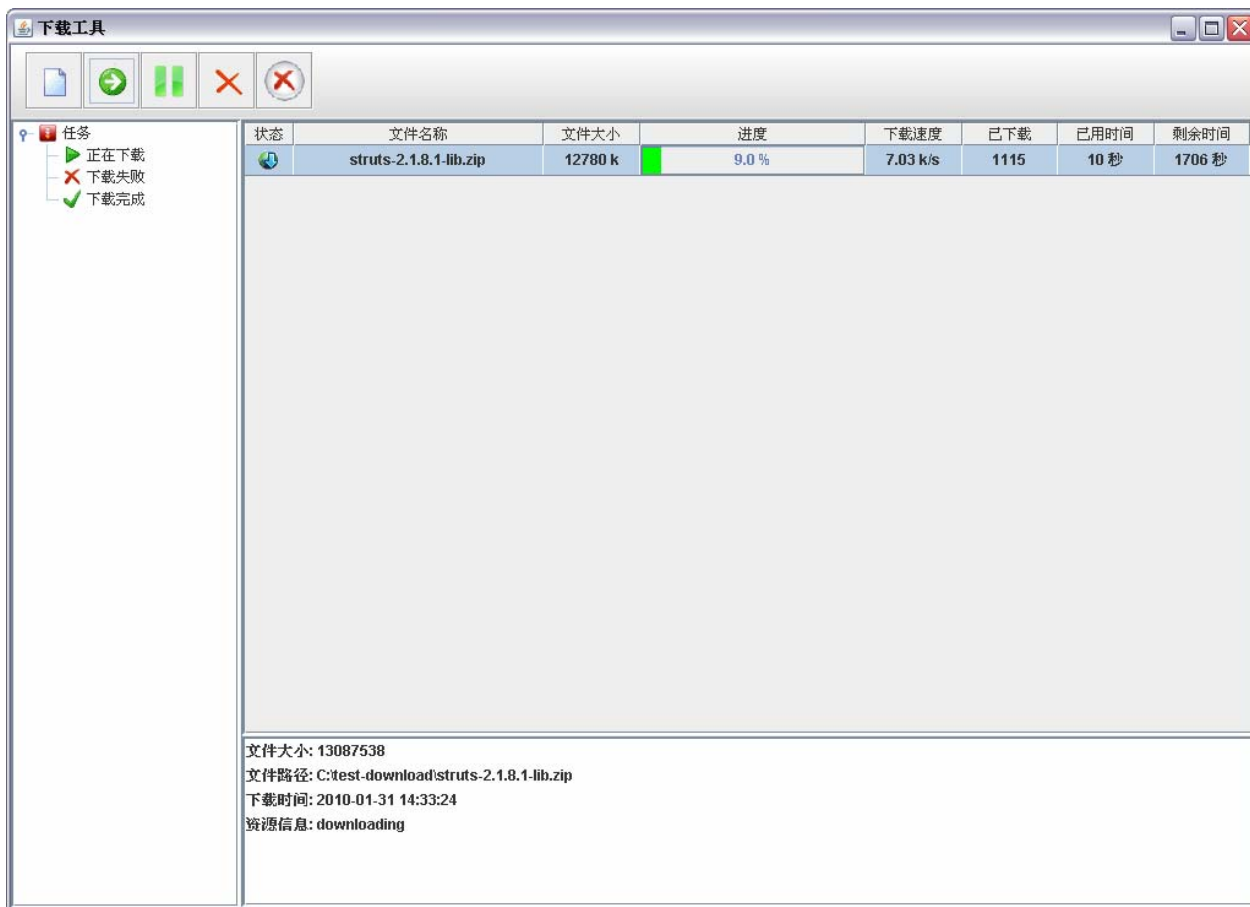


图 11.3 下载列表效果

下载列表的效果中，我们的数据都是虚拟的，如果我们以后需要加入真实的数据，只需要在 `DownloadTableModel` 中加入相应的数据集即可。

### 11.2.4 资源信息显示

其实在资源列中，我们已经可以看到资源的多数信息，资源信息显示区域可以作为对资源列表的一个补充，例如显示资源下载的绝对路径，下载时间等。在本章中，该区域只负责显示文件大小、文件保存的路径、下载时间与资源状态。我们使用一个 `JList` 对象来实现信息显示区，在 `MainFrame` 中创建该对象。

建立一个 `Info` 的类来表示 `JList` 中的一个元素。

代码清单：code\flashget\src\org\crazyit\flashget\info\Info.java:

```
public class Info {  
    //对应的值  
    private String value;  
}
```

那么在创建 `JList` 的时候，我们可以向 `JList` 中加入多个的 `Info` 对象。以下是在 `MainFrame` 中创建 `JList` 的代码。

代码清单：code\flashget\src\org\crazyit\flashget\uiMainFrame.java:

```
private void createList() {  
    this.infoJList = new JList();  
    //加入各个 Info 对象  
    this.infoList.add(this.fileSize);  
    this.infoList.add(this.filePath);  
    this.infoList.add(this.downloadDate);  
    this.infoList.add(this.info);  
    this.infoJList.setListData(infoList.toArray());  
}
```

到此，主界面的各个对象已经建立，主界面的三个区域中，只有资源列表实现较为复杂，导航树、资源信息显示都较为简单，我们在前面的案例中都提供了不同的实现。但是下载列表中并没有实际的数据，我们在下面的章节中将会实现。

### 11.2.5 新建下载任务界面

新建下载界面主要是让用户新建一个下载资源，一个新的下载资源包括资源下载地址、保存目录和保存的文件名，另外，由于本章编写的是一个多线程下载工具，因此还要提供下拉框让用户选择下载的线程数。新建界面的效果如图 11.4 所示。

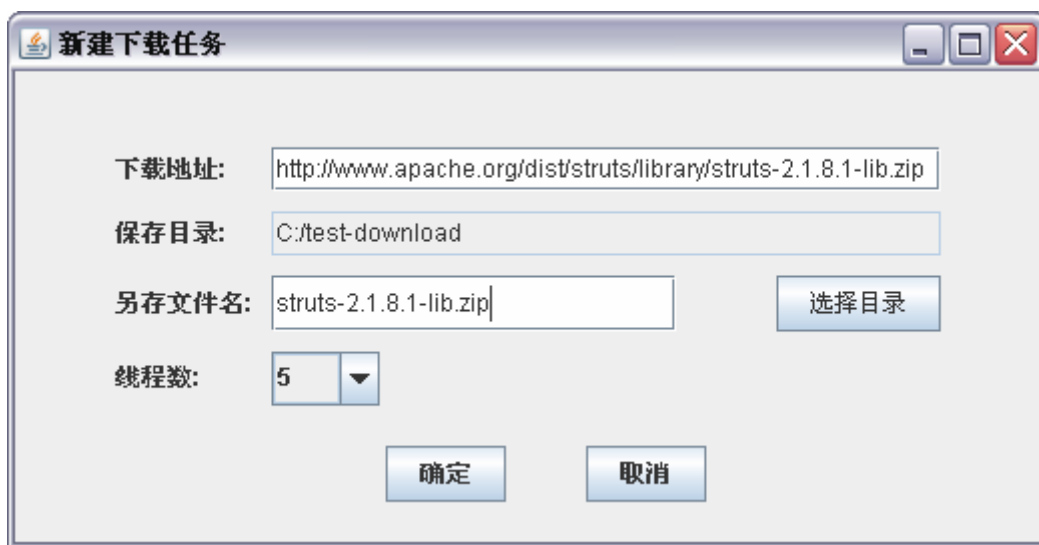


图 11.4 新建下载界面

新建下载界面实现较为简单，只是一些普通的界面控件，其他需要注意的是，当用户输入了下载地址的时候，我们需要编写程序截取相应的文件名，作为“另存文件名”的值，当用户最终点击确定的时候，我们就需要从“下载地址”与“另存文件名”中得到最终的文件名。如果不能得到文件名，则进行相关的提示。

## 11.2.6 悬浮窗口

实现一个悬浮窗口，可以继承 `Swing` 中的 `JWindow` 类，使用 `setSize` 方法来设置该 `JWindow` 对象的大小，并重写父类的 `paint` 方法，将相应的图片“画”到 `JWindow` 中。

代码清单：`code\flashget\src\org\crazyit\flashget\ui\SuspendWindow.java`：

```
public SuspendWindow(MainFrame mainFrame) {
    this.mainFrame = mainFrame;
    //得到屏幕大小
    Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();
    //设置窗口大小
    this.setSize(36, 36);
    //设置窗口位置
    int locationX = screen.width - screen.width / 5;
    int locationY = screen.height - (int)(screen.height / 1.06);
    this.setLocation(locationX, locationY);
    //设置该窗口总是在前
    this.setAlwaysOnTop(true);
    this.setVisible(true);
}

public void paint(Graphics g) {
    g.drawImage(img, 0, 0, this);
}
```

悬浮窗口的具体效果如图 11.5 所示。

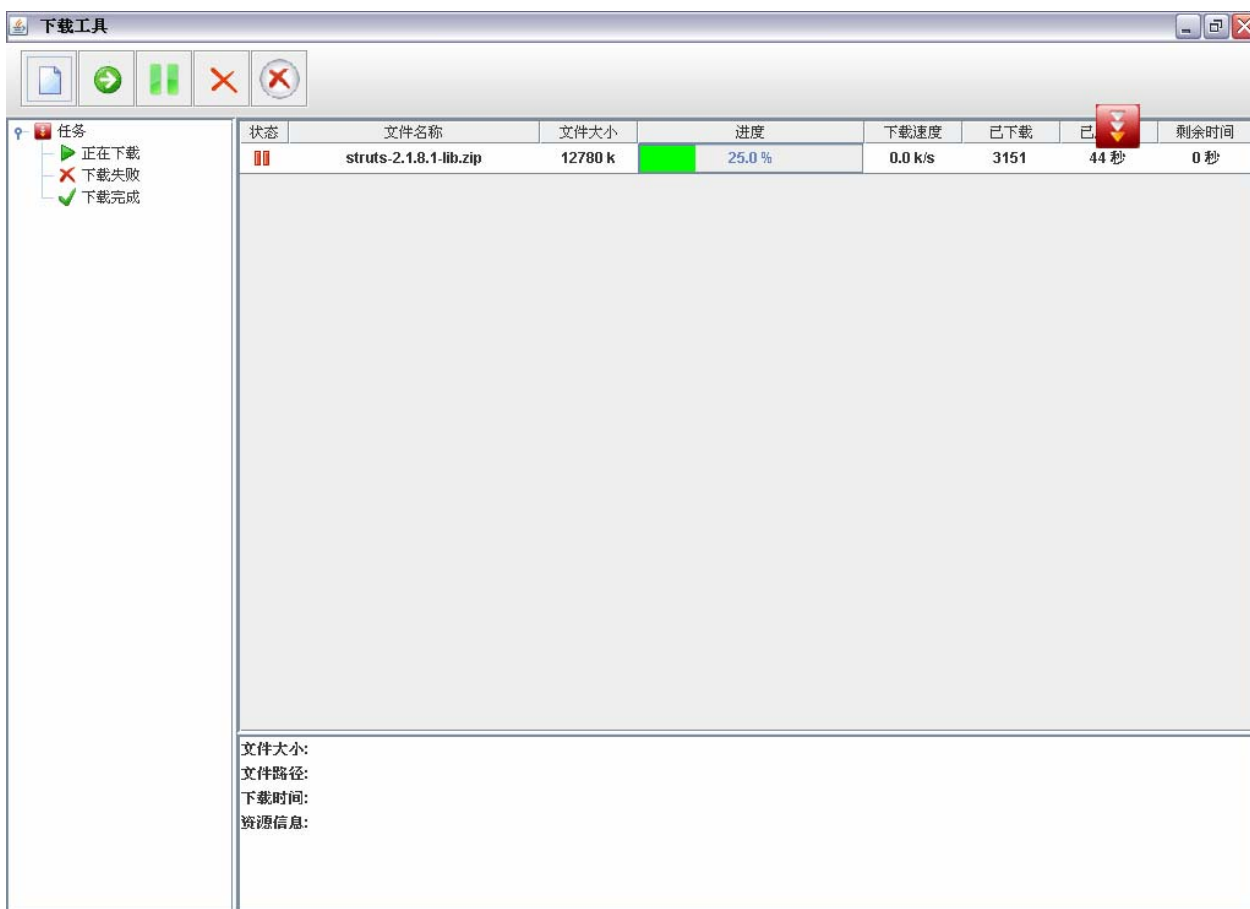


图 11.5 悬浮窗口

建立了悬浮窗口后，我们还要为该窗口加入相关的右键菜单，我们可以使用 `JPopupMenu` 类来实现鼠标的右键菜单，`JPopupMenu` 中存放各个 `JMenuItem`。

代码清单：code\flashget\src\org\crazyit\flashget\ui\SuspendWindow.java:

```
private void createPopupMenu() {
    //加入菜单
    this.popupMenu.add(this.openItem);
    ...
}
```

创建了菜单后，我们需要为窗口加入鼠标监听器，当用户进行右键点击的时候，就要将菜单显示。

代码清单：code\flashget\src\org\crazyit\flashget\ui\SuspendWindow.java:

```
this.addMouseListener(new MouseAdapter() {
    public void mouseReleased(MouseEvent e) {
        if (e.isPopupTrigger()) {
            popupMenu.show(SuspendWindow.this, e.getX(), e.getY());
        }
    }
});
```

以上代码的黑体部分显示右键菜单。在 `JWindow` 中使用 `JPopupMenu` 类的时候，如果当前的 Java 版本是 1.6.0 时，那么右键菜单将会不能出现，这是该版本的一个 Bug，如果读者使用的是 1.6.0 版本，那么建议使用更高版本的 JDK。

为悬浮窗口加入了右键菜单后，还需要实现悬浮窗口的拖动。我们只要在窗口对象中保存窗口的坐标属性，当用户对窗口进行拖动的时候再对窗口的位置进行相应的改变。

代码清单: code\flashget\src\org\crazyit\flashget\ui\SuspendWindow.java:

```
this.addMouseMotionListener(new MouseMotionAdapter() {
    public void mouseDragged(MouseEvent e) {
        //获得当前鼠标在屏幕中的坐标
        int xScreen = e.getXOnScreen();
        int yScreen = e.getYOnScreen();
        setLocation(xScreen - x, yScreen - y);
    }
});
this.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        x = e.getX();
        y = e.getY();
    }
})
```

以上代码中为鼠标加入了 `MouseMotionListener` 监听器, 实现了 `mouseDragged` 方法, 当用户在悬浮窗口进行点击的时候, 就通过以上的黑体代码来设计坐标值, 当进行拖动的时候, 就会调用 `JWindow` 的 `setLocation` 方法进行位置的改变。实现了这些后, 悬浮窗口的最终效果如图 11.6 所示。

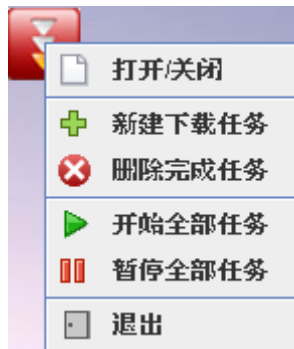


图 11.6 悬浮窗口最终效果

我们已经创建了悬浮窗口的菜单, 这些菜单的具体实现在下载的章节中将会描述。

### 11.2.7 任务栏图标

JDK1.6 提供了系统托盘（任务栏）的支持, 我们可以使用 `TrayIcon` 类来表示一个任务栏的图标, 使用该类可以为任务栏的图标加入菜单、提示文本等信息, 以下代码创建任务图标。

代码清单: code\flashget\src\org\crazyit\flashget\ui>MainFrame.java:

```
SystemTray tray = SystemTray.getSystemTray();
this.trayIcon = new TrayIcon(trayIconImage, "多线程下载工具", this.popupMenu);
this.trayIcon.setToolTip("多线程下载工具");
tray.add(this.trayIcon);
```

由于 `TrayIcon` 只支持使用 `awt` 的 `PopupMenu` 作为鼠标右键弹出菜单, 因此我们在这里没有使用 `Swing` 的 `JPopupMenu` 作为弹出菜单。加入任务栏后的效果如图 11.7 所示。



图 11.7 加入系统任务栏

到此, 多线程下载工具的界面全部创建, 这些界面组件实现相对较为简单, 接下来, 我们将设计下载的相关对象。

## 11.3 设计下载的相关对象

在实现下载功能前，我们需要设计相关的对象，例如要下载某个资源，我们可以将这个资源抽象成一个具体的对象，该对象有许多的状态（下载中、暂停、下载完成等），那么我们需要为这些状态创建相应的对象。由于本章实现的是一个多线程下载工具，因此一个资源将会被分为多个块进行下载，所以还需要创建一个“块”对象。

### 11.3.1 资源状态对象

一个资源在我们的下载工具有多种状态，包括下载中、下载完成、暂停和下载失败等，我们新建一个 **TaskState** 接口来表示资源的状态，**TaskState** 的方法如下。

代码清单：code\flashget\src\org\crazyit\flashget\state\TaskState.java:

```
/**
 * 返回该状态下的图片
 * @return
 */
ImageIcon getIcon();
/**
 * 返回状态的字符串
 * @return
 */
String getState();
/**
 * 该状态初始化执行的方法
 */
void init(Resource resource);

/**
 * 该状态结束时执行的方法
 */
void destory(Resource resouse);
```

**TaskState** 接口中有 **getIcon** 方法，该方法返回状态的图片，我们在 11.2.3 中创建了资源列表，资源列表中有一列资源状态，该列中将会显示相应的图片来表示当前资源的状态，因此在 **TaskState** 中需要提供一个返回状态图标的方法。**TaskState** 的 **getState** 方法用于返回资源状态的字符串。

**TaskState** 中有 **init** 与 **destory** 方法，如果一个资源被设置为某种状态，需要执行 **init** 方法，如果该资源被设置为另外的状态，那么在设置前，就需要执行 **destory** 方法，表示该状态销毁。

在本章中，一个资源可以有多种状态，因此需要为每一个状态加入不同的实现。由于并不是每一种状态都需要实现 **init** 与 **destory** 方法，所以在接口与实现类中加入一个抽象类（**AbstractState**），让该抽象类去实现 **init** 与 **destory** 方法（空实现），那么它的子类只需要实现 **getIcon** 与 **getState** 方法即可，如果有需要，则再去实现 **init** 与 **destory** 方法。

新建一个 **Downloading** 类，继承 **AbstractState**，该类表示正在下载的状态。该类返回相应的状态图标与状态文本。

代码清单：code\flashget\src\org\crazyit\flashget\state\Downloading.java:

```
public ImageIcon getIcon() {
    return ImageUtil.DOWNLOADING_IMAGE;
}
public String getState() {
```

```

        return "downloading";
    }

```

使用同样的方法创建其他状态类，在本章中，资源的状态包括：正在进行连接（**Connecting**）、正在下载（**Downloading**）、下载失败（**Failed**）、下载完成（**Finished**）和暂停下载（**Pause**）。

### 11.3.2 块对象

本章中实现的是一个多线程下载工具，在开始下载的时候，我们需要将一个资源拆分成若干的块，再执行若干条线程，每一条线程负责下载自己的块，因此我们需要建立一个块的对象，来保存相关的下载信息。一个块表示的下载信息有，该块的下载开始位置、该块需要下载的长度、当前下载的长度以及该块的文件名。

当下载工具得到某个资源的时候，我就需要将这个资源拆分成若干个块。例如，现在有一个大小为 100 的资源，用户选择了使用 5 条线程进行下载（见图 11.4，选择下载线程），那么就意味着要将这个资源分拆成 5 份，每一份大小为 20，那么第一块的开始下载位置就是 0，下载长度是 20，当前下载长度是 0，其他块依此类推，如果线程在某个时段下载了 10，那么当前下载长度就为 10。本章中使用一个 **Part** 对象来表示一个分拆的块。

代码清单：code\flashget\src\org\crazyit\flashget\object\Part.java:

```

//下载的开始位置
private int begin;
//这个 part 文件的长度
private int length;
//已经下载的文件长度
private int currentLength;
//每个 Part 对应的文件
private String partName;
public Part(int begin, int length, int currentLength) {
    this.begin = begin;
    this.length = length;
    this.currentLength = currentLength;
    this.partName = UUID.randomUUID() + ".part";
}

```

以上代码表示一个块，其中需要注意的是，开始下载的位置、下载长度与当前的下载位置都可以由外界提供，但是一个块对象的名称，就需要由自己进行创建，以上代码的黑体部分，使用 **UUID** 来生成唯一的文件名，而且文件后缀为 **.part**。

### 11.3.3 资源对象

当用户输入了下载地址等下载信息后，一个资源对象就要存在于我们的系统中，该资源对象包括了资源的所有信息，包括标识资源的 **id**、连接地址、保存目录、文件名称、资源大小等一系列相关的属性。在本章中使用一个 **Resource** 对象表示一个资源。

代码清单：code\flashget\src\org\crazyit\flashget\object\Resource.java:

```

//标识该资源的 id
private String id;
//资源地址
private String url;
//资源保存路径
private String filePath;
//资源下载后的名称

```

```
private String fileName;  
//资源名称  
private String sourceName;  
//资源下载后的文件对象  
private File saveFile;  
//状态  
private TaskState state;  
//文件大小  
private int size = -1;  
//下载日期  
private Date downloadDate;  
//进度  
private float progress;  
//下载速度  
private float speed;  
//使用的时间  
private int costTime;  
//剩下的时间  
private int spareTime;  
//文件所拆分的块  
private List<Part> parts;  
//下载的线程数  
private int threadSize;  
//上一次下载的大小  
private int preLength;  
public Resource(String url, String filePath, String fileName, int threadSize) {  
    this.id = UUID.randomUUID().toString();  
    this.url = url;  
    this.filePath = filePath;  
    this.fileName = fileName;  
    this.parts = new ArrayList<Part>();  
    this.saveFile = new File(filePath + File.separator + fileName);  
    this.state = DownloadContext.CONNECTION;  
    this.threadSize = threadSize;  
}
```

以上是 **Resource** 对象的属性与构造器，其中该对象的构造器只有字符串 **url**、保存目录、文件名称与线程数，即添加下载任务界面（图 11.4）的各个界面组件值，其他的属性由构造器进行设置。其中 **parts** 属性是一个集合对象，表示这个资源需要拆分的块。当一个资源进行构造的时候，也就是用户输入了下载信息点击确定的时候，就需要将这个资源的状态设置为连接中（**Connecting**），系统应该马上去进行连接，如果可以正常连接，则需要马上进行下载。

在这里需要注意的是，如果要得到文件的大小（**size** 属性），需要对资源进行一次连接，对资源进行连接使用 **URLConnection** 类。本章实现的是 **HTTP** 下载，因此需要用到 **URLConnection** 类，**URLConnection** 可以生成 **HTTP** 请求，我们在实现下载的时候，同样也需要该类。以下是获得资源大小的方法实现。

代码清单：code\flashget\src\org\crazyit\flashget\object\Resource.java:

```
public int getSize() {  
    //进行一次文件连接  
    URL resourceURL = new URL(this.url);  
    //判断之前是否已经取过文件大小
```



```

        if (this.size == -1) {
            //创建 HTTP 连接对象
            HttpURLConnection urlConnection = (HttpURLConnection)resourceURL.openConnection();
            urlConnection.connect();
            //得到资源大小
            this.size = urlConnection.getContentLength();
            //取得文件大小后返回
            urlConnection.disconnect();
        }
        return this.size;
    }
}

```

以上的代码中，为了提升性能，需要判断 **size** 属性的值，如果该属性已经被赋值，那么将不会再进行连接，这避免了每次调用 **getSize** 方法的时候，再去对资源进行一次连接。这里需要注意的是，当得到资源的大小后，需要调用 **HttpURLConnection** 的 **disconnect** 方法关闭连接。

除了 **size** 属性外，**progress** 属性（下载进度）、**speed** 属性（下载速度）和 **spareTime** 属性（剩余时间）同样也是需要由 **Resource** 类进行计算，另外，我们还需要得到该资源已经下载的长度。由于这是一个多线程下载工具，每一条线程都会向一个“块”写入下载的信息，那么一个资源已下载的长度，则是全部块已下载的长度总和。以下是 **Resource** 返回这几个属性的实现。

代码清单：code\flashget\src\org\crazyit\flashget\object\Resource.java:

```

//返回下载的进度
public float getProgress() {
    this.progress = Math.round(100.0f * getCurrentLength() / getSize());
    return progress;
}
//返回下载速度，需要得到全部已下载的长度
public float getSpeed() {
    //得到当前所有块下载的大小
    int currentLength = getCurrentLength();
    //将当前下载的长度减去前一次下载的长度，得到总下载量并计算出速度
    speed = (currentLength - preLength) / 1024.0f;
    //将当前下载的长度设置为前一次下载的长度(本次速度计算已经完成)
    preLength = currentLength;
    speed = Math.round(speed * 100) / 100.0f;
    return speed;
}
//得到该资源已经下载的长度(计算所有块的长度)
public int getCurrentLength() {
    int result = 0;
    for (Part p : this.parts) {
        result += p.getCurrentLength();
    }
    return result;
}
//得到剩余时间
public int getSpareTime() {
    //得到剩余长度
    int spareSize = getSize() - getCurrentLength();
    if (this.speed == 0) return this.spareTime;
    return (spareSize / (int)this.speed) / 1000;
}
}

```

以上代码的 `getCurrentLength` 方法，计算所有块的已下载总和。`getSpeed` 方法计算下载的速度，通过 `Resource` 对象的 `preLength` 属性来保存前一次下载量，再使用本次的下载量减去前一次的下载量来计算当前的下载速度。

`Resource` 对象中有一个 `state` 属性来表示资源的状态，本章中资源的状态使用了 `TaskState` 接口来表示，并且该接口都有相应状态的实现类，在 `TaskState` 中我们定义了 `init` 与 `destory` 方法，表示状态的初始化与销毁。那么我们需要在 `Resource` 对象的 `setState` 中调用 `init` 与 `destory` 方法。

代码清单：code\flashget\src\org\crazyit\flashget\object\Resource.java:

```
public void setState(TaskState state) {
    if (this.state != null) {
        //判断参数的状态与本对象的状态是否一致
        if (!this.state.equals(state)) {
            //两个状态一致，不需要进行任何动作，不一致，执行该状态的方法
            //执行原来状态的销毁方法
            this.state.destory(this);
            //执行新状态的 init 方法
            state.init(this);
        }
    }
    this.state = state;
}
```

以上代码的黑体部分，先判断资源是否被设置了不同状态，再执行原来状态的 `destory` 方法，最后执行新状态的 `init` 方法。

### 11.3.4 上下文对象

我们需要为下载工具建立一个上下文对象，该对象保存一些公共的信息，例如资源集合、资源的状态、最大线程组等信息。

代码清单：code\flashget\src\org\crazyit\flashget\DownloadContext.java:

```
//最大线程数
public static final int MAX_THREAD_COUNT = 5;
//下载资源状态的各个实现类
public static Connecting CONNECTION = new Connecting();
public static Downloading DOWNLOADING = new Downloading();
public static Failed FAILED = new Failed();
public static Finished FINISHED = new Finished();
public static Pause PAUSE = new Pause();
//当前下载程序中所有的任务
public List<Resource> resources = new ArrayList<Resource>();
```

`DownloadContext` 中保存了资源集合，因此如果我们需要得到某种状态的资源时，就需要对这个资源集合进行过滤，以下各方法返回各种状态的资源。

代码清单：code\flashget\src\org\crazyit\flashget\DownloadContext.java:

```
public List<Resource> getFaileds() {
    return getResources(FAILED);
}
public List<Resource> getDownloadings() {
    return getResources(DOWNLOADING);
}
public List<Resource> getFinisheds() {
    return getResources(FINISHED);
}
```

```

    }
    private List<Resource> getResources(TaskState state) {
        List<Resource> result = new ArrayList<Resource>();
        //遍历资源集合, 判断相应的状态
        for (Resource r : resources) {
            if (state.getState().equals(r.getState().getState())) {
                result.add(r);
            }
        }
        return result;
    }
}

```

如果要得到正在下载的资源, 调用 `DownloadContext` 中的 `getDownloadings` 方法即可。通过以上代码, 我们可以看到, `DownloadContext` 的各个方法都不是静态的, 也就是意味着要使用 `DownloadContext` 的方法, 就必须实例化这个类。为了保证这个类只有一个实例, 可以建立一个 `ContextHolder` 的类来保存 `DownloadContext` 的唯一实例。

代码清单: `code\flashget\src\org\crazyit\flashget\ContextHolder.java`:

```

public class ContextHolder {
    //下载工具上下文
    public static DownloadContext ctx = new DownloadContext();
}

```

那么如果需要使用 `DownloadContext` 的各个方法时, 可以使用 `ContextHolder.ctx.XXX` 语句。

到此, 下载工具的各个对象已经设计完成, 下面的章节中, 将介绍如何实现资源的下载与块的分拆。

## 11.4 下载资源

在 11.3.3 中创建了资源对象, 实现资源的下载, 只需要通过 11.2.5 中的添加资源下载界面新建一个 `Resource` 对象, 添加到上下文对象的集合中即可。新建一个 `Resource` 的时候, 我们需要根据用户选择的线程数来对下载的资源进行分块, 用户选择了 5 条线程, 那么就需要为 `Resource` 对象添加 5 个 `Part` 对象, 再为每一个 `Part` 对象启动一条下载的线程, 下载完后, 再将这 5 个 `Part` 对象放到下载的结果文件中。

### 11.4.1 界面新增下载资源

我们在 11.2.5 中建立了新建下载资源的界面, 只需要得到用户的界面输入, 再封装成一个 `Resource` 对象即可。以下是新增下载资源界面的具体实现。

代码清单: `code\flashget\src\org\crazyit\flashget\ui\NewTaskFrame.java`:

```

//返回需要保存的文件名
private String getSaveFileName() {
    String url = this.address.getText();
    String saveFileName = this.saveFileName.getText();
    //如果保存的文件名称为空, 则从 url 中截取文件名称
    if (saveFileName == null || saveFileName.equals("")) {
        saveFileName = FileUtil.getFileName(url);
    }
    return saveFileName;
}
//用户点击确定时触发的方法

```

```

private void confirm() {
    if (getSaveFileName() == null) {
        this.warnLabel.setText("请输入正确的文件名");
        return;
    }
    this.warnLabel.setText(" ");
    Resource r = createResource();
    ContextHolder.ctx.resources.add(r);
    this.setVisible(false);
}
//根据界面输入创建 Resource 对象
private Resource createResource() {
    String url = this.address.getText();
    String filePath = this.target.getText();
    String fileName = getSaveFileName();
    int threadCount = (Integer)this.threadCount.getSelectedItem();
    return new Resource(url, filePath, fileName, threadCount);
}

```

以上代码的黑体部分，创建了 **Resource** 对象后，就该对象加入到下载工具的上下文中，以上代码中的 **confirm** 方法由用户点击确定时触发。将新的资源放到上下文（**DownloadContext**）中后，那么下载列表就可以根据上下文中的资源集合，对资源列表进行渲染。

代码清单：code\flashget\src\org\crazyit\flashget\ui\DownloadTableModel.java:

```

public DownloadTableModel() {
    super();
    this.resources = ContextHolder.ctx.resources;
}
//得到记录总数
public int getRowCount() {
    if (this.resources != null) {
        return this.resources.size();
    } else {
        return 0;
    }
}
public Object getValueAt(int row, int column) {
    Resource r = this.resources.get(row);
    if (r == null) return super.getValueAt(row, column);
    String columnName = this.getColumnName(column);
    if (columnName.equals(STATE_COLUMN)) {
        return r.getState().getIcon();
    } else if (columnName.equals(FILE_NAME_COLUMN)) {
        return r.getFileName();
    } else if (columnName.equals(FILE_SIZE_COLUMN)) {
        return r.getSize() / 1024 + " k";
    } else if (columnName.equals(PLAN_COLUMN)) {
        return r.getProgress();
    } else if (columnName.equals(SPEED_COLUMN)) {
        return r.getSpeed() + " k/s";
    } else if (columnName.equals(HAS_DOWN_COLUMN)) {
        return r.getCurrentLength() / 1024;
    }
}

```

```

    } else if (columnName.equals(COST_TIME_COLUMN)) {
        return r.getCostTime() + " 秒";
    } else if (columnName.equals(SPARE_TIME_COLUMN)) {
        return r.getSpareTime() + " 秒";
    } else if (columnName.equals(ID_COLUMN)) {
        return r.getId();
    }
    return super.getValueAt(row, column);
}

```

以上代码中，为 `TableModel` 新建一个 `List<Resource>` 的属性，该 `TableModel` 对象进行构造的时候，就从上下文中得到相应的资源集合，需要注意的是，在 `TableModel` 中，我们需要为资源集合提供 `setter` 方法，由于 `TableModel` 只是负责控制界面的资源显示，它并不知道该显示哪些状态的资源，因此当用户在导航树中点击查看其他状态资源的时候，就允许外界调用 `TableModel` 的资源集合的 `setter` 方法来设置界面显示的数据。

新建一个下载资源，执行保存后效果如图 11.8 所示。

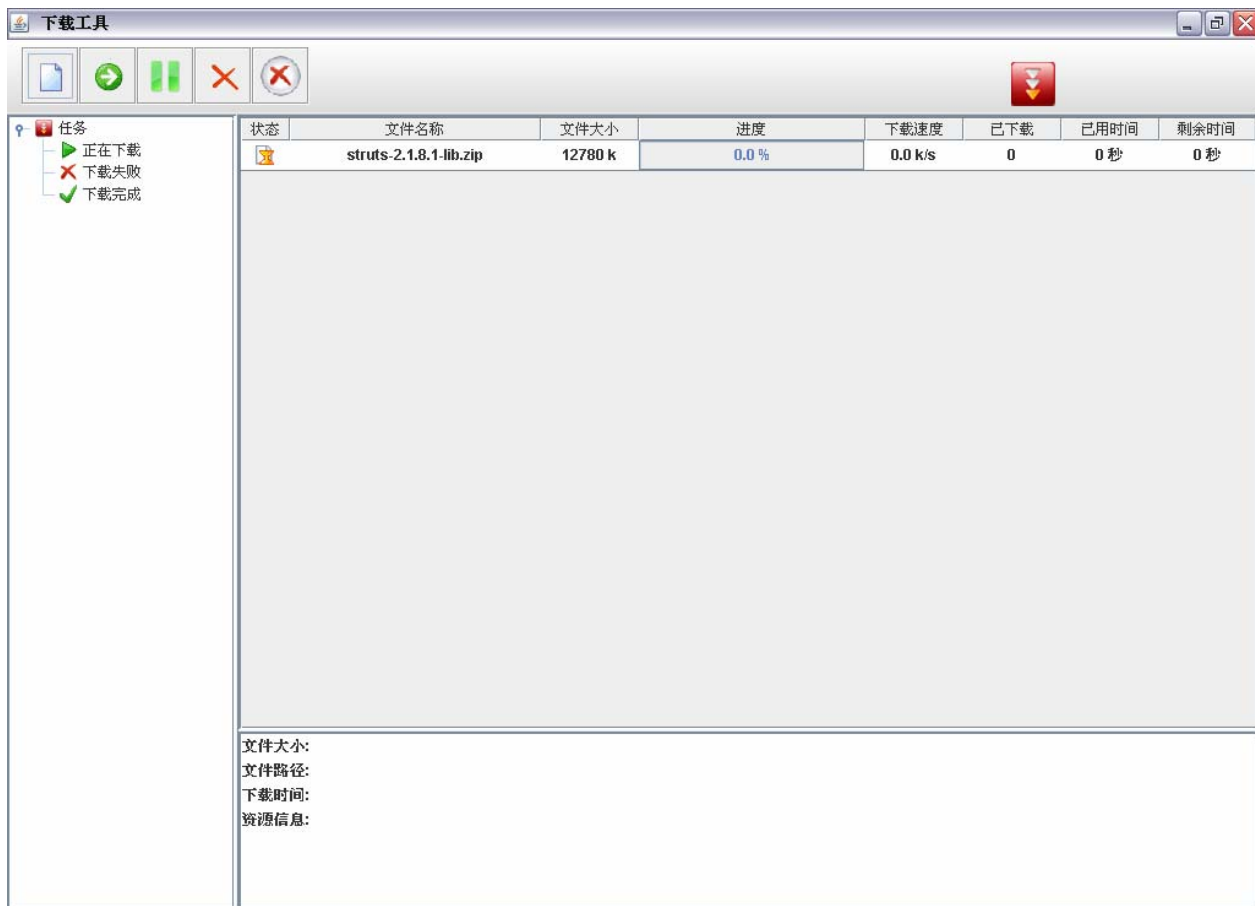


图 11.8 新增下载

`Resource` 对象在构造的时候，就设定资源状态为连接中（`Connecting`），因此图 11.8 中资源的状态为连接中状态。在 `Resource` 对象中，提供了一个 `getSize` 的方法（11.3.3），该方法用于进行 HTTP 连接并返回资源的大小，因此我们可以在图 11.8 中看到“文件大小”一列为资源的大小。

### 11.4.2 建立下载线程

用户输入了资源下载的相关信息后，就需要根据用户选择的线程数进行块的分割，再根据这些分割的块来创建相应的下载线程。建立下载线程类 `DownloadThread`，该类继承于 `Thread` 类，我们只需要实现 `run` 方法即可。如果资源被分为 5 块，那么就要创建 5 条线程，从这里可以看出，`Thread` 是与某个块相关联的，因此构造器中必须要有 `Part` 对象。

`DownloadThread`:

```
/**
 * 下载线程构造器
 */
public DownloadThread(Resource resource, RandomAccessFile raf, Part part) {
    this.url = createURL(resource.getUrl());
    this.raf = raf;
    this.part = part;
    this.resource = resource;
}
private URL createURL(String urlPath) {
    return new URL(urlPath);
}
```

实现文件的下载，主要是使用 `URLConnection` 读取远程的资源，再通使用 `RandomAccessFile` 对象将读取到子节输出到本地文件中。以下是 `DownloadThread` 的 `run` 实现。

代码清单：`code\flashget\src\org\crazyit\flashget\thread\DownloadThread.java`:

```
public void run() {
    //计算开始点与结束点
    int begin = part.getBegin() + part.getCurrentLength();
    int end = part.getBegin() + part.getLength() - 1;
    //如果是开始点大于结束点，证明该块已经下载完成
    if (begin >= end) {
        this.raf.close();
        return;
    }
    HttpURLConnection urlConnection = (HttpURLConnection)url.openConnection();
    urlConnection.setRequestProperty("Range", "bytes=" + begin + "-" + end);
    //如果连接不上相应的地址，抛出 java.net.UnknownHostException
    urlConnection.connect();
    //如果找不到相应的资源，将抛出 java.io.FileNotFoundException
    InputStream is = urlConnection.getInputStream();
    byte[] buffer = new byte[MAX_BUFFER_SIZE];
    int perRead = 0;
    //设置状态为下载
    this.resource.setState(DownloadContext.DOWNLOADING);
    //在.part 文件中设置当前所读取的指针
    this.raf.seek(this.part.getCurrentLength());
    while ((perRead = is.read(buffer)) != -1) {
        //判断资源对象状态
        raf.write(buffer, 0, perRead);
        this.part.setCurrentLength(this.part.getCurrentLength() + perRead);
    }
    closeStream(is, urlConnection, this.raf);
}
```

```

    }
    //关闭参数中的流
    private void closeStream(InputStream is, HttpURLConnection urlConnection,
        RandomAccessFile raf) throws IOException {
        is.close();
        urlConnection.disconnect();
        raf.close();
    }
}

```

在以上代码中，先计算出开始下载点与开下的结束点，如果一个资源大小是 100，被分成 5 份，那么第一个 Part 对象的开始点就是 0，结束点就是 19，长度是 20。当这个 Part 对象被下载到 10 的时候程序终止，程序就需要记录当前的下载点，通过以上代码的黑体部分记录当前 RandomAccessFile 文件的指针。当第二次进行下载的时候，我们可以得知 Part 对象的开始点是 0，结束点是 19，长度是 20，但是此时已下载属性（Part 的 `currentLength` 属性）就是 10。计算开始点的时候，需要将 Part 的 `currentLength` 属性加上开始点，让程序知道本次下载需要从 10 开始，到 19 结束。指定下载的起点与结束点，可以使用 `HttpURLConnection.setRequestProperty("Range", "bytes=" + begin + "-" + end)` 进行设置。当 `HttpURLConnection` 对象进行连接后，可以根据该对象得到一个 `InputStream` 对象，读取资源返回的流，再将 `InputStream` 读取到的字段通过 `RandomAccessFile` 的 `write` 方法写入到本地的文件中（.part 文件）。

### 11.2.3 进行文件分割

在进行文件分割前，我们建立一个 `DownloadHandler` 对象，用于处理文件的下载与继续下载，该类提供 `doDownload` 方法，用户新建下载资源时，可以调用该方法进行下载。`DownloadHandler` 同时也需要提供 `resumeDownload` 方法，当资源被停止下载后，如果想重新进行下载，可以调用 `resumeDownload` 方法，重新进行下载。

在 11.2.2 中，我们创建了下载线程，那么在实现 `doDownload` 方法的时候，就可以根据用户选择的线程数对下载的资源进行分割，创建一个 Part 对象，再启动一条下载线程。以下是 `doDownload` 方法的具体实现。

代码清单：code\flashget\src\org\crazyit\flashget\thread\DownloadHandler.java:

```

public void doDownload(Resource r) {
    //设置下载日期
    if (r.getDownloadDate() == null) r.setDownloadDate(new Date());
    r.setState(DownloadContext.CONNECTION);
    //计算出每一块的大小
    int partLength = r.getSize() / r.getThreadSize() + 1;
    //时间计算任务
    CountTimeTask timeTask = new CountTimeTask(r);
    Timer timer = new Timer();
    timer.schedule(timeTask, 0, 1000);
    for (int i = 0; i < r.getThreadSize(); i++) {
        int length = partLength;
        //如果是最后一块，则使用总数来减去前面块的总和
        if (i == (r.getThreadSize() - 1)) {
            length = r.getSize() - i * partLength;
        }
        //创建各个 Part 对象
        Part p = new Part((i * partLength), length, 0);
        r.getParts().add(p);
        RandomAccessFile raf = new RandomAccessFile(r.getFilePath() +

```

```

        File.separator + p.getPartName(), "rw");
        DownloadThread t = new DownloadThread(r, rav, p);
        //设置线程优先级
        t.setPriority(6);
        t.start();
    }
}

```

DownloadHandler 的 doDownload 方法中，只需要计算出每一块的大小，再创建一个 RandomAccessFile 对象，就可以启动下载线程。在创建 RandomAccessFile 对象的时候，我们使用 Part 对象的 partName 属性创建 .part 文件，该属性在构造 Part 的时候，就通过 UUID 生成唯一的名称。需要注意的是，一条线程只会下载一个 .part 文件。以上代码的黑体部分是一个定时器，用于计算资源所使用的时候，该定时器每隔一秒，将 Resource 的 costTime 属性加 1 即可。

当用户新建下载资源的时候，就可以调用 DownloadHandler 的 doDownload 方法进行下载，由于 DownloadHandler 不必每次使用的时候都创建一个实例，可以将其放到 ContextHolder 类中初始化。

代码清单：code\flashget\src\org\crazyit\flashget\ContextHolder.java:

```

public class ContextHolder {
    //下载工具上下文
    public static DownloadContext ctx = new DownloadContext();
    //下载处理类
    public static DownloadHandler dh = new DownloadHandler();
}

```

用户新建下载资源点击 NewTaskFrame 的确定按钮，触发发该类的 confirm 方法，只要向该方法调用 doDownload 方法进行下载即可。

代码清单：code\flashget\src\org\crazyit\flashget\ui\NewTaskFrame.java:

```

private void confirm() {
    if (getSaveFileName() == null) {
        this.warnLabel.setText("请输入正确的文件名");
        return;
    }
    this.warnLabel.setText(" ");
    Resource r = createResource();
    ContextHolder.ctx.resources.add(r);
    ContextHolder.dh.doDownload(r);
    this.setVisible(false);
}

```

以上的黑体代码调用 DownloadHandler 的 doDownload 方法进行下载。我们可以新建下载资源进行测试，可以看到系统已经帮我们生成了若干份 .part 文件，并且这些文件的大小不停在增长，说明我们的下载线程已经生效。

线程程序已经编写完了，但是主界面的资源列表并不会随着文件的下载而发生变化，这是由于我们需要为界面组件加入相关的控制，让其在某段时间内进行刷新。

代码清单：code\flashget\src\org\crazyit\flashget\ui\MainFrame.java:

```

ActionListener refreshTable = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        //刷新列表
        downloadTable.updateUI();
    }
};
public MainFrame {
    ...
}

```



```
//创建定时器
Timer timer = new Timer(1000, refreshTable);
timer.start();
...
}
```

使用 `JTable` 的 `updateUI` 方法对界面进行刷新,需要注意的是,以上代码的 `Timer` 对象是 `java.swing` 包中的 `Timer` 类。此时可以运行程序,新建下载,可以看到效果如图 11.9 所示。

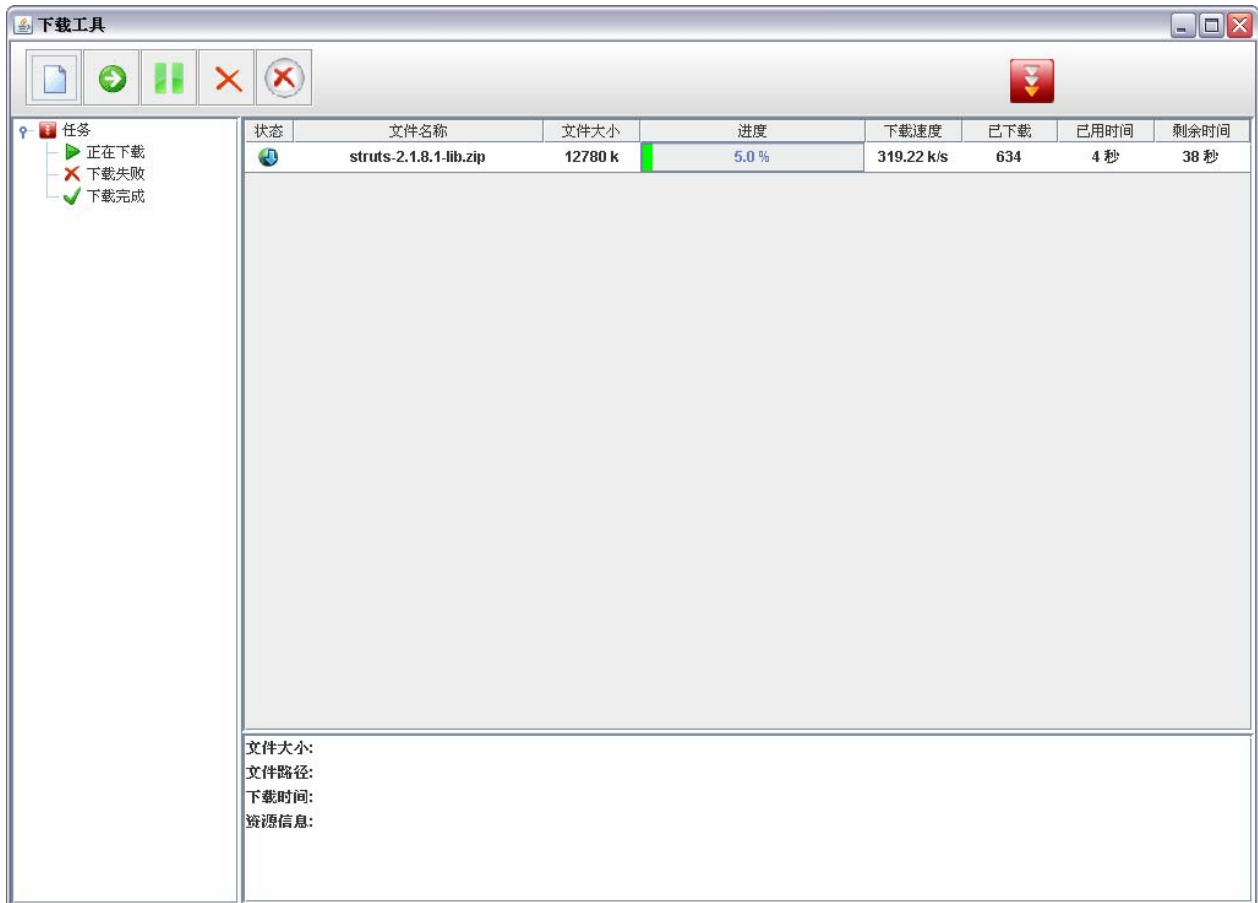


图 11.9 实现文件下载线程

### 11.2.4 文件合并

当文件下载完成后,到保存目录,可以看到,只是有几份`.part`文件在那,而我们所需要的目标文件却没有产生,这是由于下载线程只是帮我们将字节保存到这些`.part`文件中,并没有帮我们进行合并,因此我们需要在下载线程中编写文件合并的实现,并所有的`.part`文件合并成一份目标文件。在这里需要注意的是,合并前我们需要判断是否已经下载完成,而判断一个文件是否下载完成,需要判断该资源里面所有的`.part`文件是否已经下载完成,我们要得到该资源对应的所有 `Part` 对象,再用这些 `Part` 对象的 `currentLength` 属性相加,判断是否大于资源的 `size`。在本章中,一个 `Part` 对象由一条线程进行下载,那么就意味着每一条线程在下载完自己那一部分的字节后,都需要进行一次判断。

代码清单: `code\flashget\src\org\crazyit\flashget\thread\DownloadThread.java`:

```
public void run() {
    ...
    //判断是否下载完成,如果下载完成,则进行合并文件
}
```

```

        //注意这里需要得到整个文件的大小，而不是某个.part 文件的大小
        if (isFinished(this.resource.getSize())) uniteParts();
    }
    //判断是否下载完成，遍历下载文件的各个.part 文件
    private boolean isFinished(int fileLength) {
        List<Part> parts = this.resource.getParts();
        //计算已下载的总数
        int downCount = 0;
        for (Part part : parts) downCount += part.getCurrentLength();
        return (downCount >= fileLength) ? true : false;
    }
    //合并 part 文件
    private void uniteParts() throws IOException {
        List<Part> parts = this.resource.getParts();
        //创建文件输出流，输出到下载文件
        OutputStream bos = new FileOutputStream(this.resource.getSaveFile(),
            false);
        for (Part part : parts) {
            //得到.part 文件
            File partFile = new File(FileUtil.getPartFilePath(this.resource,
                part));
            //获得文件输入流
            InputStream is = new FileInputStream(partFile);
            byte[] buffer = new byte[1024];
            int bytesRead;
            int temp = 0;
            while ((bytesRead = is.read(buffer)) != -1) {
                temp += bytesRead;
                //写到文件中
                bos.write(buffer, 0, bytesRead);
            }
            is.close();
        }
        bos.close();
        this.resource.setState(DownloadContext.FINISHED);
    }
}

```

以上代码中的 `isFinished` 方法判断资源是否下载完成，如果资源下载完成后，再调用 `uniteParts` 方法，将该资源所有的 `.part` 文件合并到目标文件中。文件的全并使用 IO 流即可完成，读取 `.part` 文件，再写入，最后，将资源的状态设置为完成。这里需要注意的是，将资源的状态设置为完成状态后，需要将所有的 `.part` 文件删除，并停止资源用时的计时器。

### 11.2.5 暂停下载

当用户需要暂停一个资源的下载时，就可以将所选择的资源状态改为暂停（**Pause**）即可，但是线程并不知道我们已经做了暂停操作，因此可以在 `DownloadThread` 的 `run` 方法中（`while` 循环体）加入判断，如果资源的状态被设置成 **Pause** 的话，那么 `run` 方法就返回。

代码清单：code\flashget\src\org\crazyit\flashget\thread\DownloadThread.java:

```

public void run() {
    ...
}

```

```

while ((perRead = is.read(buffer)) != -1) {
    //判断资源对象的状态是否被修改成暂停
    if (this.resource.getState() instanceof Pause) {
        closeStream(is, urlConnection, this.raf);
        return;
    }
    ...
}
}

```

### 11.2.6 继续下载

当一个状态为下载中的资源被暂停后，用户需要让其继续下载，那么我们可以实现 `DownloadHandler` 中的 `resumeDownload` 方法。让系统得到当前资源的 `Part` 对象，再根据这些对象找到相应的 `.part` 文件并根据这些文件创建 `RandomAccessFile` 对象，最后以 `Part`、`RandomAccessFile` 和 `Resource` 重新启动一条线程。由于 `Part` 中保存了相应的读取信息（读取开始点、长度、已读取长度），那么线程就会根据这些读取信息来计算开始点和结束点，由于我们使用 `RandomAccessFile` 中的 `seek` 方法记录了文件指针，那么就可以继续进行下载。

代码清单：code\flashget\src\org\crazyit\flashget\thread\DownloadHandler.java:

```

public void resumeDownload(Resource r) {
    if (r.getState() instanceof Finished) return;
    //计算使用时间
    CountTimeTask timeTask = new CountTimeTask(r);
    Timer timer = new Timer();
    timer.schedule(timeTask, 0, 1000);
    //将 Timer 对象放到 Map 中, key 为该资源的 id
    timers.put(r.getId(), timer);
    for (int i = 0; i < r.getParts().size(); i++) {
        //得到 Part 对象
        Part p = r.getParts().get(i);
        //创建 RandomAccessFile 对象
        RandomAccessFile rav = new RandomAccessFile(r.getFilePath() +
            File.separator + p.getPartName(), "rw");
        //启动线程
        DownloadThread t = new DownloadThread(r, rav, p);
        t.start();
    }
}

```

继续下载的方法与开始下载的方法实现并无太大区别，继续下载方法中可以通过 `Resource` 对象得到相应的 `Part` 对象，而开始下载的方法则是创建 `Part` 对象。

到此，文件的下载已经全部实现，我们可以运行程序查看效果，下载完后效果如图 11.10 所示。

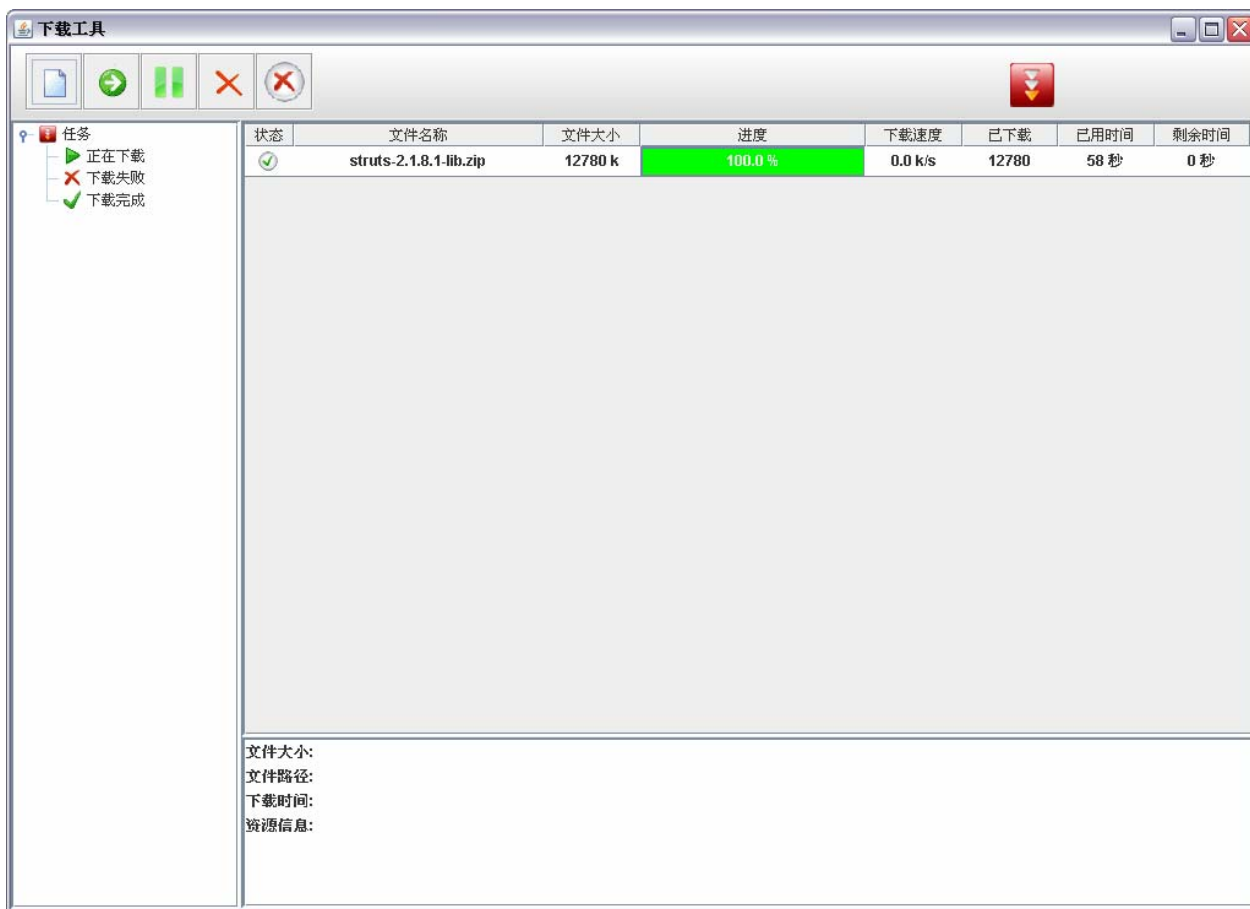


图 11.10 下载完成

## 11.5 保存下载信息

当用户关闭了下载工具，而且下载工具正在下载某些资源的时候，我们需要对这些正在下载的资源进行保存。实际上，我们需要保存的是资源对象（Resource），块对象（Part），由于资源对象保存该资源所有的信息，块对象保存了当前所下载的长度等重要信息，因此我们需要将这两个对象进行保存。在本章的程序中，我们并不涉及任何的数据库程序，因此没有数据库让我们去进行保存，因此，我们可以通过 Java 的序列化来保存所有的这些对象。

### 11.5.1 进行序列化保存对象

在开始编码前，我们需要明白要序列化哪些对象。在前面的程序中，我们创建了一个 DownloadContext 的对象，该对象保存了一个资源集合，表示当前下载工具中相关的所有资源对象，每一个资源对象中保存了所有的资源信息（速度、进度、已下载等），得到某个资源对象后，就可以得到该资源所有的块（Part 对象），就可以得到该块已下载的长度、该块的长度、开始点和结束点。因此，我们可以将 DownloadContext 进行序列化。

在 Java 中，序列化一个对象，该对象必须实现 Serializable 接口，而且各个对象中的属性必须也是可以序列化的，因此 Resource、Part 中不能出现不可以序列化的属性，在 Resource 中使用了 TaskState 作为该 Resource 的状态，因此这个状态接口必须继承 Serializable 接口。如果这些对象没有实现 Serializable 接口，那么将抛出 java.io. InvalidClassException。

为 `MainFrame` 提供一个 `serializable` 的方法，用于程序退出时进行序列化。

代码清单：code\flashget\src\org\crazyit\flashget\ui\MainFrame.java:

```
//序列化(DownloadContext 对象)
public void serializable() {
    //序列化前先将所有正在下载的任务停止
    for (Resource r : ContextHolder.ctx.resources) {
        if (r.getState() instanceof Downloading) {
            r.setState(ContextHolder.ctx.PAUSE);
        }
    }
    File serFile = FileUtil.SERIALIZABLE_FILE;
    //判断序列化文件是否存在，不存在则创建
    if (!serFile.exists()) serFile.createNewFile();
    FileOutputStream fos = new FileOutputStream(serFile);
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    //将上下文对象写到序列化文件中
    oos.writeObject(ContextHolder.ctx);
    oos.close();
    fos.close();
}
```

以上代码中，使用 `ObjectOutputStream` 将对象写到 `txt` 文件中，而这个对象就是我们的 `DownloadContext`（上下文）对象。`serializable` 方法定义在 `MainFrame` 中，但是，由于我们实现了悬浮窗口以及任务栏图标，因此该方法由悬浮窗口和任务栏图标的右键菜单调用，当程序退出的时候，调用 `MainFrame` 的 `serializable` 方法。

代码清单：code\flashget\src\org\crazyit\flashget\ui\SuspendWindow.java:

```
this.quitItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        mainFrame.serializable();
        System.exit(0);
    }
});
```

## 11.5.2 反序列化

实现了序列化将 `DownloadContext` 保存到序列化文件中后，我们还需要编写程序进行反序列化，在下载工具启动的时候，从序列化文件中得到之前的 `DownloadContext` 对象，主要是得到各个资源对象。

代码清单：code\flashget\src\org\crazyit\flashget\ui\MainFrame.java:

```
//将 DownloadContext 中的所有资源对象设置到列表的 TableModel 中
private void addTableData() {
    DownloadTableModel model = (DownloadTableModel)this.downloadTable.getModel();
    //将保存的资源设置到列表中
    model.setResources(ContextHolder.ctx.resources);
    //刷新列表
    this.downloadTable.refresh();
}
//反序列化
public void reverseSer() {
    File serFile = FileUtil.SERIALIZABLE_FILE;
```

```

if (!serFile.exists()) return;
try {
    //得到文件输入流
    FileInputStream fis = new FileInputStream(serFile);
    ObjectInputStream ois = new ObjectInputStream(fis);
    //设置 ContextHolder 的 DownloadContext
    ContextHolder.ctx = (DownloadContext)ois.readObject();
    ois.close();
    fis.close();
} catch (Exception e) {
    e.printStackTrace();
}
//设置列表
addTableData();
}

```

以上代码中的 `reverseSer` 方法进行反序列化，由于 `DownloadContext` 对象保存在 `ContextHolder` 中，因此当通过反序列化得到 `DownloadContext` 对象后，还需要将该对象设置到 `ContextHolder` 中，最后为下载列表设置数据，让列表可以显示相应的数据。我们可以在 `MainFrame` 构造器中调用 `reverseSer` 方法。到此，我们实现了用户退出程序后保存当前下载资源的信息，用户退出程序后，不必担心之前所下载的资源消失，重新启动下载工具后，就可以继续下载资源。

## 11.6 其他操作

在前面的章节中，我们实现了资源的下载、暂停、继续，程序退出对资源进行序列化后，启动程序时进行反序列化。在本章的下载工具中，还有删除任务、删除已经完成任务、开始全部任务、暂停全部任务和查看任务等功能没有实现，那么在本小节，将会实现这些操作。

### 11.6.1 查看任务

当用户在下载列表选择了某一个资源时，就可以将该资源的相关信息添加到信息显示区的 `JList` 中。实现较为简单，只需要为下载列表加入鼠标监听器即可。

代码清单：`code\flashget\src\org\crazyit\flashget\ui\MainFrame.java`:

```

//点击列表鼠标监听器
this.downloadTable.addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent e) {
        //得到点击的资源
        Resource r = getResource();
        if (r == null) return;
        //设置信息显示区域的值
        fileSize.setValue(FILE_SIZE_TEXT + r.getSize());
        filePath.setValue(FILE_PATH_TEXT +
            r.getSaveFile().getAbsolutePath());
        downloadDate.setValue(DOWNLOAD_DATE_TEXT +
            DateUtil.formatDate(r.getDownloadDate()));
        info.setValue(RESOURCE_INFO_TEXT + r.getState().getState());
        //重新设置 JList 数据
        infoJList.setListData(infoList.toArray());
    }
});

```

```
}
});
```

查看资源效果如图 11.11 所示。

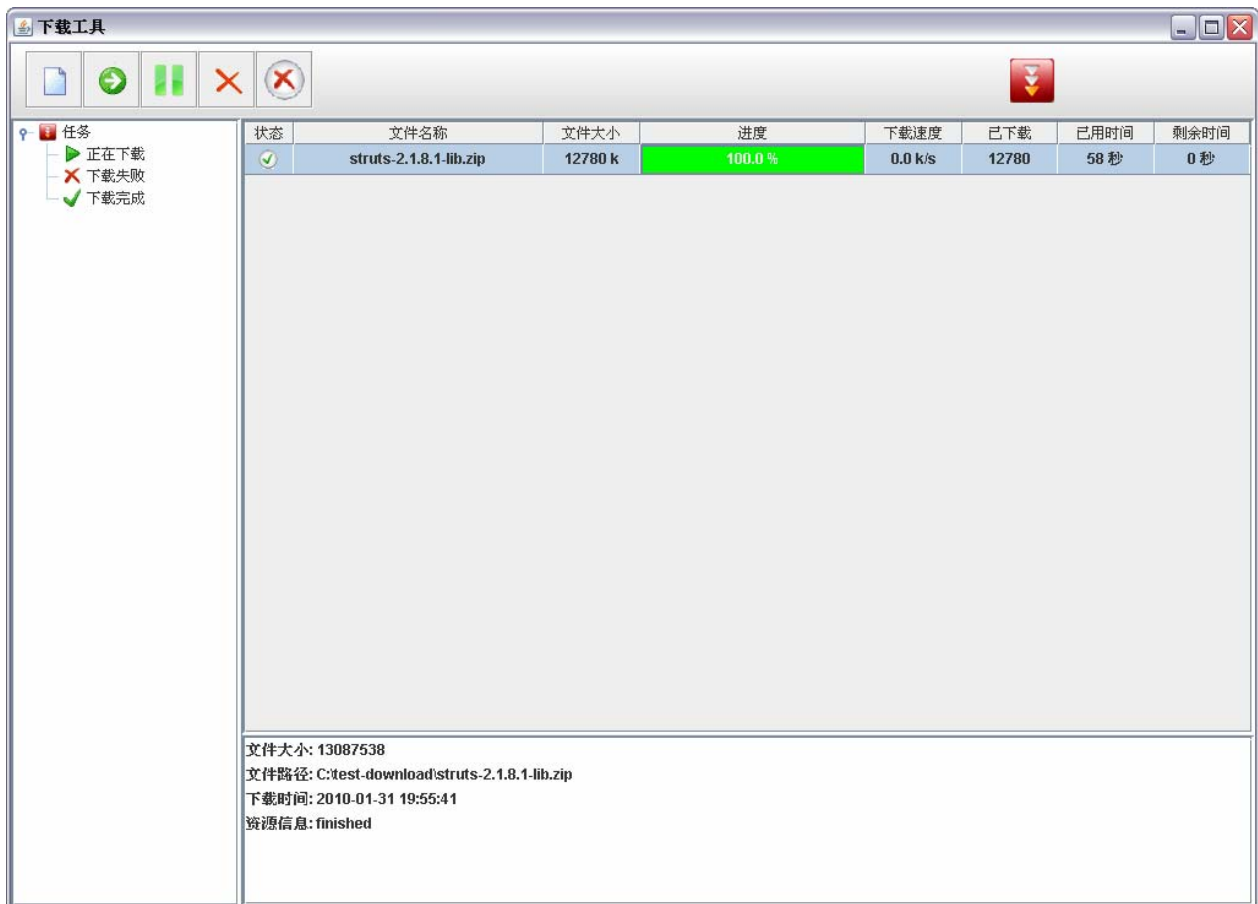


图 11.11 查看资源

### 11.6.2 删除任务

删除一个任务，需要先将任务暂停，再删除该资源相应的.part 文件。由于资源在下载时，每一个.part 文件都有一条线程对其进行数据写入，如果不将任务暂停，那么将不可以删除这些.part 文件，由于这些.part 文件正在被使用。我们在实现下载线程的时候，在 while 循环体中加入了判断，如果一个资源被暂停，那么就会关闭这个资源所有流，因此需要删除一个任务并删除它所对应的.part 文件，就要先将该任务的状态设置为暂停状态，再从上下文（DownloadContext）的资源集合中删除该资源。

代码清单：code\flashget\src\org\crazyit\flashget\uiMainFrame.java:

```
/**
 * 删除资源
 */
private void delete() {
    Resource r = getResource();
    if (r == null) return;
    //先将任务停止
    r.setState(ContextHolder.ctx.PAUSE);
    //删除所有的.part 文件
```

```
FileUtil.deletePartFiles(r);
//从上下文集合中删除资源
ContextHolder.ctx.resources.remove(r);
}
```

### 11.6.3 删除已经完成的任務

删除已经完成的任務比较简单，由于这些资源已经下载完成，因此不会存在.part 文件，只需要将这个资源从上下文中删除即可。

代码清单：code\flashget\src\org\crazyit\flashget\ui\MainFrame.java:

```
/**
 * 删除已下载完成的资源
 */
public void deleteFinished() {
    for (Iterator it = ContextHolder.ctx.resources.iterator(); it.hasNext();) {
        Resource r = (Resource)it.next();
        if (r.getState() instanceof Finished) {
            it.remove();
        }
    }
}
```

注意：以上的 deleteFinished 方法是 public 的，这是由于悬浮窗口与任务栏都有删除完成任务的功能。

### 11.6.4 开始全部任务

开始全部任务功能只在悬浮窗口和任务栏图标的鼠标右键菜单中出现，我们可以在 MainFrame 中提供一个 public 的方法，用于开始全部的任务，根据上下文得到全部的资源，并对这些资源进行判断，如果资源状态为 Pause（暂停）或者 Failed（下载失败），那么就可以开始（将资源的状态设置为 Downloading），如果为其他状态（正在下载或者下载完成），则不可以开始。

代码清单：code\flashget\src\org\crazyit\flashget\ui\MainFrame.java:

```
/**
 * 开始全部任务
 */
public void startAllTask() {
    for (Resource r : ContextHolder.ctx.resources) {
        if (r.getState() instanceof Pause || r.getState() instanceof Failed) {
            ContextHolder.dh.resumeDownload(r);
        }
    }
}
```

### 11.6.5 暂停全部任务

实现暂停全部任务与实现开始全部任务类似，只是将正在下载的任务全部暂停（设置正在下载任务的状态为 Pause）。

MainFrame:



```

/**
 * 暂停全部任务
 */
public void pauseAllTask() {
    for (Resource r : ContextHolder.ctx.resources) {
        if (r.getState() instanceof Downloading) {
            r.setState(ContextHolder.ctx.PAUSE);
        }
    }
}

```

### 11.6.6 节点的点击

用户在导航树中点击了某个节点的时候，就需要根据所点击的节点，得到相应的资源集合。在创建上下文对象（DownloadContext）的时候，就已经为该对象创建了返回各个状态资源的方法，我们只需要为导航树加入鼠标监听器，再调用上下文对象的相应方法即可实现。

代码清单：code\flashget\src\org\crazyit\flashget\uiMainFrame.java:

```

private void initlisteners() {
    ...
    //点击导航树鼠标监听器
    this.navTree.addMouseListener(new MouseAdapter(){
        public void mouseClicked(MouseEvent e) {
            selectTree();
        }
    });
    ...
}
/**
 * 点击导航树触发的方法
 */
private void selectTree() {
    DownloadNode selectNode = getSelectNode();
    this.currentNode = selectNode;
    refreshTable();
}
/**
 * 刷新列表
 */
private void refreshTable() {
    DownloadTableModel model = (DownloadTableModel)this.downloadTable.getModel();
    model.setResources(ContextHolder.ctx.getResources(currentNode));
}

```

以上代码，根据用户所选择的节点设置 MainFrame 的 currentNode 属性，再根据该属性去刷新下载列表，就可以得到相应的资源数据。

### 11.6.7 打开/关闭主窗口

用户对悬浮窗口进行鼠标双击的时候，如果主窗口（MainFrame）是隐藏的，那么就需要将主窗口

(MainFrame) 显示出来 (setVisible(true))。用户点击了悬浮窗口菜单 (如图 11.6 所示) 的“打开/关闭”时, 就需要显示或者隐藏主窗口。

代码清单: code\flashget\src\org\crazyit\flashget\ui\SuspendWindow.java:

```
this.openItem.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent arg0) {  
        if (mainFrame.isVisible()) {  
            mainFrame.setVisible(false);  
        } else {  
            mainFrame.setVisible(true);  
        }  
    }  
});
```

到此, 多线程下载工具已经全部实现, 我们可以从网上找一些资源进行相关的测试。

## 11.7 本章小结

本章中实现了一个 HTTP 多线程下载工具, 学习了使用 `HttpURLConnection` 进行资源连接、下载, 并使用 `RandomAccessFile` 来实现断点续传的功能, 使用了序列化与反序列化保存对象信息。界面中使用了自定义的 `TableModel` 来动态更新列表数据, 实现类似于 `Flashget`、迅雷的悬浮窗口, 并将程序加入到系统任务栏图标中。本章的重点是多线程下载, 在程序中将一个资源分割为若干个“块”, 为每一个“块”建立一条线程进行下载, 实现了多线程下载, 下载完所有的“块”后, 再使用 IO 流将这些“块”合并到最终的文件中。本章的开发使用的 Java 版本是 1.6.0\_18, 操作系统是 Windows XP, 本章的程序可以在该环境下成功运行。希望本章的多线程下载工具, 对读者能有很大启发。

## 第 12 章 邮件客户端

在日常的生活中，我们都使用到许多的邮件客户端，例如 **Foxmail**、**Outlook** 等，这些邮件客户给我们收发邮件带来了方便，不必再去打开网页去查看邮件，只打开这些客户端，就可以轻松的收发邮件。本章我们将介绍如何使用 **Java** 去开发一款自己的邮件客户端，功能并不需要太复杂，可以收发邮件，管理邮件即可，从这些功能中体会这些客户端的原理。

### 12.1 本章涉及的技术

使用 **Java** 开发邮件客户端，首先想到的就是 **JavaMail API**，另外，我们还需要对邮件进行保存，例如像 **Foxmail** 一样将邮件下载到本地的系统中，使用一些文件来对这些邮件进行保存，这些做可以提高邮件客户端的性能，不必每次都上互联网上去下载这些邮件，只需要通过操作本地系统中的文件就可以达到操作邮件的目的，为了满足这个要求，我们还会使用 **XStream** 这个项目，下面先对 **JavaMail** 进行简单的描述。

#### 12.1.1 JavaMail简介

**JavaMail API** 是 **Sun** 提供的处理电子邮件的 **API**，可以方便的使用它来进行一些常用的邮件操作，它提供了独立于各个平台和独立于各种协议的框架，可以让我们去构建一些邮件和消息的应用。我们可以从以下网址得到 **JavaMail** 的包：

<http://java.sun.com/products/javamail/downloads/index.html>

本章中所使用的 **JavaMail** 版本为 **1.4.2**，下载了 **JavaMail** 后，将 **mail.jar** 加入项目的 **CLASS\_PATH** 中，就可以使用 **JavaMail API**。

本章中使用了 **SMTP** 协议和 **POP3** 协议来进行发送邮件和接收邮件，下面介绍这两种协议。

#### 12.1.2 SMTP协议简介

**SMTP** 是 **Simple Mail Transfer Protocol** 的缩写，译为简单邮件传输协议，**SMTP** 被用来在互联网上传递各种电子邮件的协议，可以将邮件从本客户端传送到邮件服务器，简单的说，我们可以使用这个协议发送邮件。在一般的情况下，**SMTP** 使用 **25** 作监听端口。大部分的邮箱都支持这 **SMTP** 协议，例如 **Gmail** 的邮箱提供的 **SMTP** 服务器地址是：**smtp.gmail.com**，如果我们需要使用我们的 **Gmail** 来发送邮件，就需要使用到这一个 **SMTP** 服务器地址。

#### 12.1.3 POP3 协议简介

**POP3** 是 **Post Office Protocol** 的缩写，即邮局协议，用于电子邮件的接收，一般情况下，它使用 **TCP** 的 **110** 端口，由于现在最常用的是第三版，因此称为 **POP3**。简单的说，这个协议用于接收电子邮件，因此我们在本章的邮件客户端中，将会使用这个协议来接收邮件。

### 12.1.4 XStream简介

XStream 是一个简单的 Java 库，它可以将 Java 对象转换成 XML，也可以将 XML 转换成 Java 对象。在本章中，我们从远程的邮件服务器得到邮件后，需要下载到本地进行保存，再将远程邮件服务器中的邮件删除，这样当我们浏览邮件的时候，可以不必再从远程的邮件服务器下载，提高了邮件客户端的性能。由于本章中不涉及数据库应用，因此可以将邮件转换成一些 XML 格式的文件，保存在本地系统中，每次操作邮件的时候，都可以对这些 XML 文件进行操作，提高了邮件客户端的性能。

我们可以从下面的网址得到 XStream：

<http://xstream.codehaus.org/download.html>

本章中所使用的版本为 1.3.1，下载后将 lib 下的 xstream-1.3.1.jar 包与 xpp3\_min-1.1.4c.jar 的包加到项目的 CLASS\_PATH 中即可使用 XStream 的 API。JavaMail 和 XStream 的使用将在以下章节介绍如何使用。

## 12.2 建立界面

在开发邮件的主要功能前，我们先建立相关的界面。我们需要先确定要些什么界面，首先，当然是需要一个进入系统的界面（登录），让用户去输入用户名，根据用户名去建立与这个用户名相关的用户信息。接下来，需要建立一个主界面，用来进行查看邮件、发送邮件、接收邮件和处理邮件等操作，然后，还需要有一个配置界面，让用户去配置邮箱（用户名和密码）、SMTP 协议和 POP3 协议等信息，最后，需要提供一个写邮件的界面，让用户去输入邮件信息，并发送到具体的地址。

### 12.2.1 登录界面

我们需要提供一个登录界面，让用户确定用户名，我们这里并不需要进行密码验证，因为用户能否连入他的邮件，取决于对邮箱的配置。登录界面如图 12.1 所示。



图 12.1 登录界面

登录界面十分简单，一个输入框和两个按钮就构成了登录界面，点击取消就退出客户端。该界面对应的 Java 类为项目中的 LoginFrame。

### 12.2.2 客户端主界面

客户端的主界面需要进行各种的邮件处理，因此界面相对较为复杂，主界面的最终效果如图 12.2 所示。

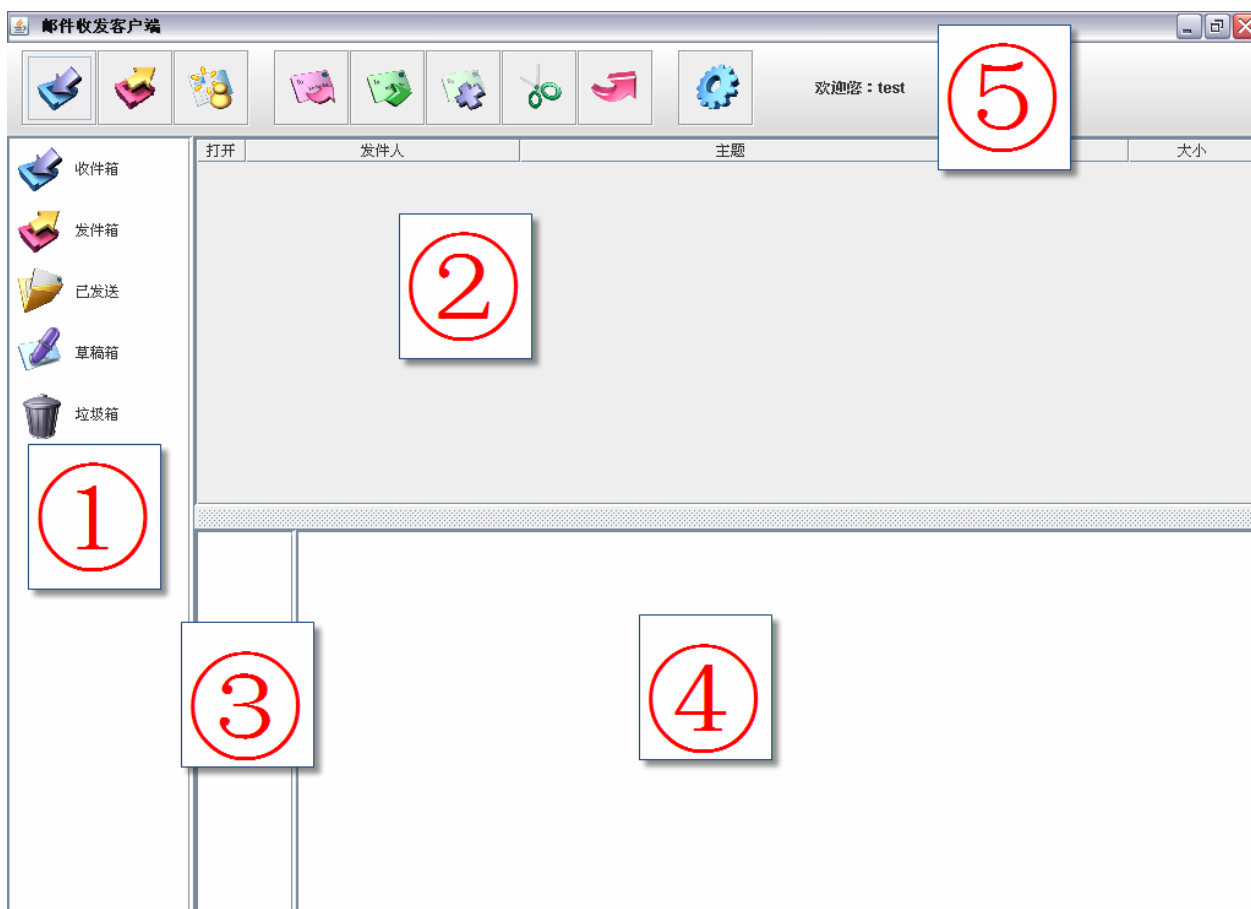


图 12.2 客户端主界面

图 12.2 中客户端的主界面中结构较为复杂，图 12.2 中的①区域，是一棵导航树，点击不同的链接，可以进入到收件箱、发件箱、已发送、草稿箱和垃圾箱中，当我们点击导航树上的各个节点时，②区域的列表就显示不同类型的邮件，当我们点击了②区域列表中的某一封邮件时，邮件的主要信息就会显示在④区域，该邮件的附件就会显示在③区域，⑤区域是一个工具条，存放对邮件的各个操作，该工具条所定义操作分别有（图 12.2 的区域⑤由左往右）：

- ☐ 收取邮件：到远程的邮件服务器中收取新收的邮件。
- ☐ 发送邮件：当用户选择了某一份在发件箱、已发送、草稿箱和垃圾箱的邮件时，就弹出写邮件的界面，将选择的邮件信息显示到写邮件的界面。
- ☐ 写邮件：弹出写邮件的界面，让用户编写邮件。
- ☐ 回复邮件：当用户选择了某一份邮件点击回复时，弹出写邮件的界面。
- ☐ 转发邮件：与回复邮件一样，但是转发时收信人为空。
- ☐ 删除邮件：将所选择的邮件放到垃圾箱中。
- ☐ 彻底删除邮件：在本系统中彻底删除该邮件，一旦删除，将不可被还原。
- ☐ 还原邮件：将在垃圾箱中的邮件还原到原来的目录。
- ☐ 设置：配置用户邮箱相关的信息，包括账号、密码、SMTP 协议与 POP3 协议等。

客户端主界面对应的类为 **MainFrame**。实现图 12.2 的界面，需要注意的是导航树的实现，由于在外观上这并不是一棵树，但在本章中使用了树，以下是导航树的实现代码，**MainFrame** 的 **createTree** 方法。

代码清单：code\foxmail\src\org\crazyit\foxmail\uiMainFrame.java

```

//创建导航的树
private JTree createTree() {
    //创建根节点
    DefaultMutableTreeNode root = new DefaultMutableTreeNode();
    //加入各个子节点
    root.add(new DefaultMutableTreeNode(new Inbox()));
    root.add(new DefaultMutableTreeNode(new OutBox()));
    root.add(new DefaultMutableTreeNode(new SentBox()));
    root.add(new DefaultMutableTreeNode(new DraftBox()));
    root.add(new DefaultMutableTreeNode(new DeletedBox()));
    //创建树
    JTree tree = new JTree(root);
    //加入鼠标监听器
    tree.addMouseListener(new SailTreeListener(this));
    //隐藏根节点
    tree.setRootVisible(false);
    //设置节点处理类
    SailTreeCellRenderer cellRenderer = new SailTreeCellRenderer();
    tree.setCellRenderer(cellRenderer);
    return tree;
}

```

以上的代码中，使用了五个 Box 的类：Inbox、OutBox、SentBox、DraftBox 和 DeletedBox，这五个类分别代码着导航树中的五个不同链接。另外还有一个鼠标监听器类和节点处理类，以下为五个 Box 类的实现。

代码清单：code\foxmail\src\org\crazyit\foxmail\box\MailBox.java

```

public interface MailBox {
    //获得 box 名字
    String getText();
    //返回对应图标
    ImageIcon getImagelcon();
}

```

再新建一个接象类，给五个 Box 类去继承。

代码清单：code\foxmail\src\org\crazyit\foxmail\box\AbstractBox.java

```

public abstract class AbstractBox implements MailBox {
    //该 box 所对应的图片(显示在树上的图片)
    private ImageIcon icon;
    //实现接口的方法
    public ImageIcon getImagelcon(String imagePath) {
        if (this.icon == null) {
            this.icon = new ImageIcon(imagePath);
        }
        return this.icon;
    }
    //重写 toString 方法，调用接口的 getText 方法，getText 方法由子类去实现
    public String toString() {
        return getText();
    }
}

```

五个 Box 类只需要去继承 AbstractBox 并实现 getText 方法即可，以下为 Inbox 类的实现。

代码清单：code\foxmail\src\org\crazyit\foxmail\box\InBox.java

```
public class InBox extends AbstractBox {
    public String getText() {
        return "收件箱";
    }
    public ImageIcon getImagelcon() {
        return super.getImagelcon("images/in-tree.gif");
    }
}
```

InBox 类代表树中的收件箱节点，只需要返回文字与图片即可。其他的四个 Box 类与 InBox 实现一样，这里不贴出代码，具体可看本书所附的代码。

实现了五个 Box 后，我们可以在树中使用这五个 Box，以下是导航树的鼠标事件监听器类。

代码清单：code\foxmail\src\org\crazyit\foxmail\ui\SailTreeListener.java

```
public class SailTreeListener extends MouseAdapter {
    private MainFrame mainFrame;
    public SailTreeListener(MainFrame mainFrame) {
        this.mainFrame = mainFrame;
    }
    public void mousePressed(MouseEvent e) {
        //暂时提供空实现
    }
}
```

一个简单的监听器类，鼠标的点击方法没有实现，我们在下面章节会提供实现。以下代码是树的节点处理类 SailTreeCellRenderer。

代码清单：code\foxmail\src\org\crazyit\foxmail\ui\SailTreeCellRendererer.java

```
public class SailTreeCellRenderer extends DefaultTreeCellRenderer {
    //树节点被选中时的字体
    private Font selectFont;
    public SailTreeCellRenderer() {
        this.selectFont = new Font(null, Font.BOLD, 12);
    }
    public Component getTreeCellRendererComponent(JTree tree, Object value,
        boolean sel, boolean expanded, boolean leaf, int row,
        boolean hasFocus) {
        DefaultMutableTreeNode node = (DefaultMutableTreeNode)value;
        MailBox box = (MailBox)node.getUserObject();
        this.setText(box.getText());
        //判断是否选中，再决定使用字体
        if (isSelected(node, tree)) this.setFont(this.selectFont);
        else this.setFont(null);
        //设置树节点所显示的图标
        this.setIcon(box.getImagelcon());
        return this;
    }
    //判断一个 node 是否被选中
    private boolean isSelected(DefaultMutableTreeNode node, JTree tree) {
        //得到选中的 TreePath
        TreePath treePath = tree.getSelectionPath();
        if (treePath == null) return false;
        //得到被选中的节点
        DefaultMutableTreeNode selectNode = (DefaultMutableTreeNode)treePath.getLastPathComponent();
```

```
//如果选中的节点与参数的节点一致, 那么意味着参数的节点被选中
if (node.equals(selectNode)) return true;
return false;
}
}
```

由于我们在创建树节点的时候, 使用的是五个 **Box** 作为 **DefaultMutableTreeNode** 的构造参数, 因此, 以上代码的黑体部分, 我们可以得到这些 **Box** 对象, 而五个 **Box** 对象都是 **MailBox** 接口的实现, 因此只需要强制类型转换为 **MailBox** 即可, 根据得到的节点对应的 **MailBox** 对象, 可以获得该 **Box** 对应的图片与文字, 再设置到树的节点中, 最后判断节点是否被选中, 来设置不同的字体。

以上为图 12.2 中①区域导航树的实现, 整个界面的布局可以使用 **JSplitPane** 来实现, ②区域是一个 **JTable**, ③区域是一个 **JList** 对象, ④区域是一个 **JTextArea** 对象并不可编辑, ⑤区域是一个 **JToolBar** 对象, ②③④⑤区域的实现在这里不再详细描述, 请看代码清单的 `code\foxmail\src\org\crazyit\foxmail\ui\MainFrame.java`。

### 12.2.3 配置界面

配置界面主要用来配置用户的邮箱信息, 包括邮箱账号、密码、SMTP 服务器、SMTP 端口、POP3 服务器和 POP3 端口, 配置界面如图 12.3 所示。



图 12.3 配置界面

该界面并没有多复杂的结构, 只是由几个普通的 **JLabel** 与 **JTextField** 组成, 当用户填写完相关的信息并点击确定后, 配置界面将关闭, 一旦配置完后, 整个客户端将会以这些配置为依据, 去收取邮件或者发送邮件。在本章中, 配置界面对应的类为 **SetupFrame**。以下为在客户端主界面 (**MainFrame**) 打开配置界面的代码。

代码清单: `code\foxmail\src\org\crazyit\foxmail\ui\MainFrame.java`

```
//设置
private Action setup = new AbstractAction("设置", new ImageIcon("images/setup.gif")) {
    public void actionPerformed(ActionEvent e) {
        setup();
    }
};
//设置方法, 打开设置界面
private void setup() {
    if (this.setupFrame == null) {
```



```
        this.setupFrame = new SetupFrame(this);
    }
    this.setupFrame.setVisible(true);
}
```

## 12.2.4 邮件编写界面

最后来完成整个客户端的最后一个界面：编写邮件界面。邮件编写界面并不复杂，只是提供一些 `JTextField` 和一个 `JTextArea` 来让用户输入相关的邮件信息，例如收件人、抄送、邮件主题、邮件内容与附件，存放附件可以提供一个 `JList` 来实现。邮件编写界面如图 12.4 所示。

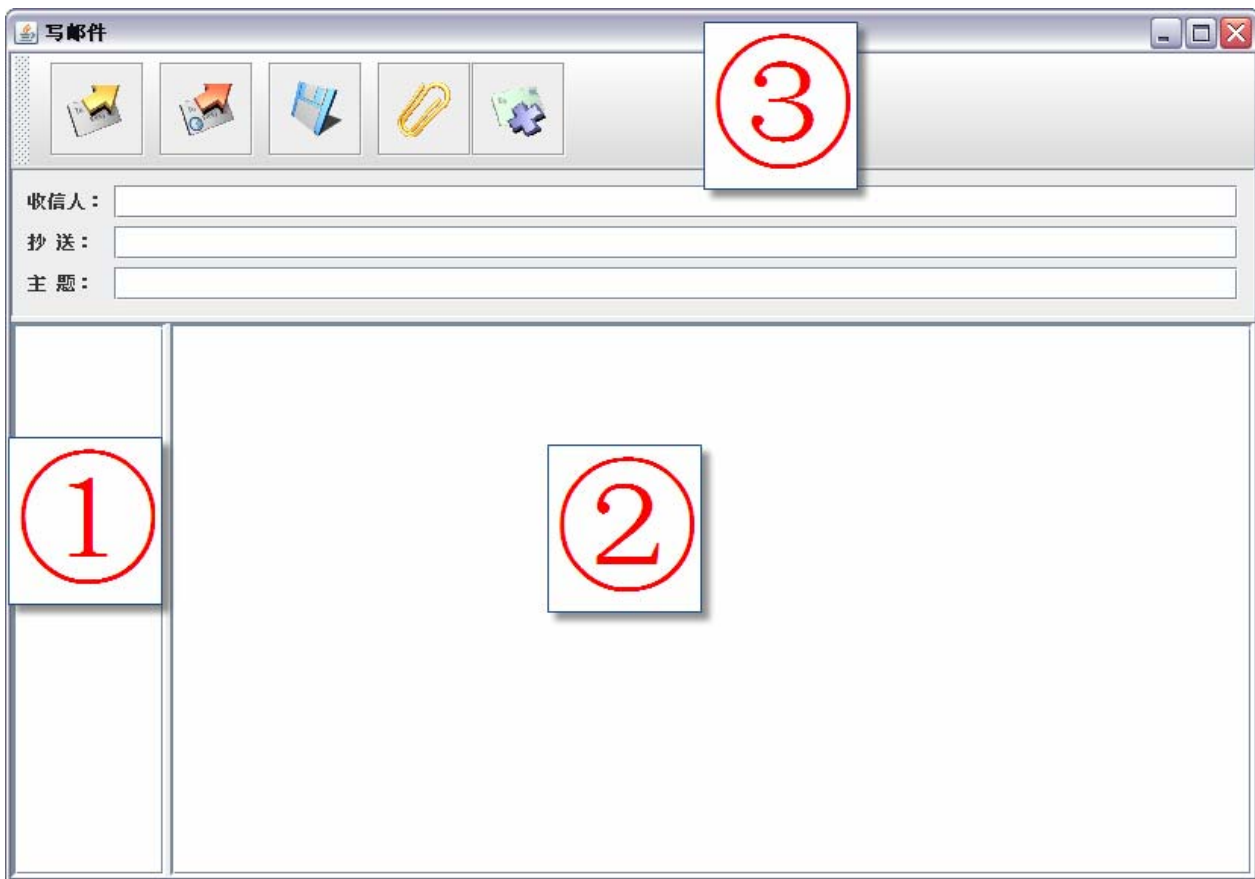


图 12.4 邮件编写界面

图 12.4 中的区域①是邮件的附件存放区域，对应一个 `JList` 对象，区域②是邮件的正文区域，区域③是一个 `JToolBar`，可以对该封正在编写的邮件进行一系列的操作，从左到右的操作分别为：

- ☐ 发送：将邮件马上发送到收件人中。
- ☐ 保存至发件箱：将该封正在编写的邮件保存到发件箱中，等待下次操作再进行发送。
- ☐ 保存至草稿箱：将该封邮件保存到草稿箱中，可以作其他邮件的模板。
- ☐ 增加附件：为正在编写的邮件增加一个附件，增加完后将在图 12.4 的区域①（`JList`）中显示。
- ☐ 删除附件：当用户在附件的列表中（`JList`）选择了一份附件点击删除时，该附件并从附件列表中删除。

邮件编写界面在本章中对应的类为 `MailFrame`，整个界面的布局都可以使用 `JSplitPane` 来实现，具体实现方式与主界面（`MainFrame`）的实现类似，而且具体的操作也与主界面相似，当用户点击了编写

邮件的时候，就需要显示该界面。以下为在主界面（MainFrame）中打开邮件编写界面（MailFrame）的代码。

代码清单：code\foxmail\src\org\crazyit\foxmail\ui\MainFrame.java

```
//写邮件
private Action write = new AbstractAction("写邮件", new ImageIcon("images/new.gif")) {
    public void actionPerformed(ActionEvent e) {
        write();
    }
};
private void write() {
    //编写邮件的界面对象 mailFrame
    this.mailFrame.setVisible(true);
}
```

到此，我们这个邮件客户端的所有界面都已经完成了，接下来我们开始编写这个邮件客户端的主体功能，包括保存用户的配置、接收邮件、发送邮件和操作邮件等功能。

## 12.3 创建客户端的基础对象

接收邮件我们使用 POP3 协议，通过用户配置的账号和密码，到远程的邮件服务器中下载邮件，并显示在收件箱的列表中，在做这些事，我们需要将邮件的各个对象先准备好，例如与邮箱相关的邮件上。除此之外还需要一个邮件的读取接口，用于读取邮件，由于我们使用 JavaMail 从邮件服务器得到的邮件对象是 Message，因此该接口需要将 Message 转换成 Mail 对象。

### 12.3.1 创建邮箱上下文对象

首先需要创建一个邮件上下文对象，用于保存邮条箱的连接信息，包括账号、密码、SMTP 协议、SMTP 端口、POP3 协议和 POP3 端口，新建 MailContext 类，该类在下面将会十分常用。

代码清单：code\foxmail\src\org\crazyit\foxmail\ui\MailContext.java

```
public class MailContext {
    //系统用户
    private String user;
    //用户帐号
    private String account;
    //密码
    private String password;
    //smtp 邮件服务器
    private String smtpHost;
    //smtp 端口
    private int smtpPort;
    //pop3 邮件服务器
    private String pop3Host;
    //pop3 的端口
    private int pop3Port;
    //是否进行重置信息
    private boolean reset = false;
    //省略各个属性的 setter 和 getter 方法
}
```

除了需要提供以上代码的属性外，还需要提供一个 **JavaMail** 的 **Store** 对象，当外界请求这个类的时候，可以马上返回这个 **Store** 对象，该对象代表邮箱的仓储。

代码清单：code\foxmail\src\org\crazyit\foxmail\ui\MailContext.java

```
private Store store;
//返回 Store 对象
public Store getStore() {
    //重置了信息，设置 Store 为 null
    if (this.reset) {
        this.store = null;
        this.session = null;
        this.reset = false;
    }
    if (this.store == null || !this.store.isConnected()) {
        try {
            Properties props = System.getProperties();
            //创建 mail 的 Session
            Session session = Session.getDefaultInstance(props, getAuthenticator());
            //使用 pop3 协议接收邮件
            URLName url = new URLName("pop3", getPop3Host(), getPop3Port(), null,
                getAccount(), getPassword());
            //得到邮箱的存储对象
            Store store = session.getStore(url);
            store.connect();
            this.store = store;
        } catch (Exception e) {
            e.printStackTrace();
            throw new MailConnectionException("连接邮箱异常，请检查配置");
        }
    }
    return this.store;
}
//将账号与密码封装成一个 JavaMail 的 Authenticator 对象，表示需要经过身份验证
private Authenticator getAuthenticator() {
    return new Authenticator() {
        protected PasswordAuthentication getPasswordAuthentication() {
            return new PasswordAuthentication(getAccount(), getPassword());
        }
    };
}
```

在这里外界可以通过一个 **getStore** 的方法得到 **Store** 对象，而且需要经过判断，如果当 **Store** 对象为空或者没有连接上的时候，就进行 **Store** 对象的创建，这是由于 **Store** 对象创建需要连接到远程的邮件服务器，因此创建或者连接都需要等待较长时间，而我们一旦配置好了之后，就无需每次都去创建，共用一个 **Store** 对象即可，这就好像 **JDBC** 中的 **Connection** 对象的创建一样，创建一次 **Connection** 系统的开销都很大，只需要一次创建，即可一直使用该实例。另外，我们还提供了一个 **reset** 的布尔值，表示该 **MailContext** 对象是否被更改过，如果用户重新设置了邮箱账号、密码等信息，那么就需要告诉 **MailContext**，重新获得 **Store** 对象。以上代码中的黑体部分是使用 **JavaMail** 去连接邮箱。

### 12.3.2 创建邮件对象

在我们的客户端中，需要提供一个 **Mail** 对象，在本系统中一个 **Mail** 对象代表一封邮件，**Mail** 对象的各个属性如下。

代码清单：code\foxmail\src\org\crazyit\foxmail\object\Mail.java

```
public class Mail {  
    //在本地系统中代表该邮件的 xml 文件的名称，使用 UUID 作为文件名：uuid.xml  
    private String xmlName;  
    //发送人  
    private String sender;  
    //收件人，可以多个  
    private List<String> receivers;  
    //邮件标题  
    private String subject;  
    //日期  
    private Date receiveDate;  
    //邮件大小  
    private String size;  
    //是否已经被阅读  
    private boolean hasRead;  
    //邮件正文  
    private String content;  
    //抄送  
    private List<String> ccs;  
    //附件  
    private List<FileObject> files;  
    //邮件来源，如果是从邮箱收到的，则放在 INBOX，其他情况对应的放  
    private String from;  
    //省略 setter 和 getter 方法  
}
```

按照我们最初的要求，如果从远程的邮件服务器下载了邮件后，需要将邮件转换成一份 XML 文件，因此这里的 **Mail** 对象就是我们需要转换的对象，那么我们就需要为 **Mail** 对象提供一个 **xmlName** 的属性，该属性代表该 **Mail** 对象所对应的 XML 文件的文件名，设置 **Mail** 的 **xmlName** 的属性时，该文件名必须是唯一的，因此需要使用 **UUID** 来生成。这个 **xmlName** 只是一个文件名，并不是 XML 文件的绝对路径。

在以上的代码中，**Mail** 对象还有一个 **files** 属性，代表该邮件所对应的所有附件，每一个附件在客户端中代表一个 **FileObject** 对象，以下是 **FileObject** 的属性。

代码清单：code\foxmail\src\org\crazyit\foxmail\object\FileObject.java

```
public class FileObject {  
    //源文件名字  
    private String sourceName;  
    //对应的文件  
    private File file;  
    //省略 setter 和 getter 方法  
}
```

**FileObject** 只保存一个 **sourceName** 和一个 **File** 对象，**sourceName** 表示用户在上传附件时的源文件的名称，当我们将文件保存在本地的时候，需要将附件重命名并以 **UUID** 作为这些文件的名称，这样做的话原来文件的文件名将会丢失，因此需要使用 **sourceName** 来保存源文件的名称。

以上为两个在本章中最基础的两个对象，下面开始实现登录功能和用户配置功能。

## 12.4 用户登录与用户配置

在 12.2.1 和 12.2.3 中，我们提供了一个登录界面和一个配置的界面，当用户进行登录时，我们需要为这个用户名创建一个具体的目录，用来保存这个用户的信息，例如该用户所使用的配置、具体使用的邮箱等，如果一个用户使用多个不同的邮箱，那么就需要在用户对应的目录下再创建多个不同的账号目录，下面先来实现用户登录的功能。

### 12.4.1 实现用户登录

在本章中，用户登录并不需要进行密码验证，只需要用户去确认自己的用户名，系统根据这个用户去创建或者定位到属于该用户的目录。以下为 `LoginFrame` 的确定方法。

代码清单：code\foxmail\src\org\crazyit\foxmail\ui>LoginFrame.java

```
private void confirm() {
    //得到用户名，可以在此加入验证，如果为空，则提示并返回
    String user = this.userField.getText();

    //得到用户名对应的目录
    File folder = new File(FileUtil.DATE_FOLDER + user);
    //该用户的目录不存在，代表该用户第一次进入系统，创建用户目录
    if (!folder.exists()) folder.mkdir();
    //得到配置文件
    File config = new File(folder.getAbsolutePath() + FileUtil.CONFIG_FILE);
    try {
        //没有对应的配置文件，则创建
        if (!config.exists()) config.createNewFile();
        //读取配置并转换为 MailContext 对象
        MailContext ctx = PropertiesUtil.createContext(config);
        //设置 MailContext 的 user 属性
        ctx.setUser(user);
        //创建系统界面主对象
        this.mainFrame = new MainFrame(ctx);
        //显示主界面
        this.mainFrame.setVisible(true);
        //隐藏登录界面
        this.setVisible(false);
    } catch (IOException e) {
        throw new LoginException("配置文件错误");
    }
}
```

在以上的代码中，用户点输入用户名并点击确定进行登录，就执行以上的 `confirm` 方法，先判断该用户是否第一次登录，判断标准为该用户对应的目录是否存在，再获得该用户的配置文件，再将配置文件里面的内容（`MailContext` 中的属性）转换成 `MailContext` 对象，再显示主界面对象。代码中使用了 `PropertiesUtil` 类进行创建 `MailContext` 对象，同时也使用了 `FileUtil` 的几个常量，以下是 `PropertiesUtil` 的实现。

代码清单：code\foxmail\src\org\crazyit\foxmail\util\PropertiesUtil.java

```
public class PropertiesUtil {
    //根据文件得到 Properties 对象
    private static Properties getProperties(File propertyFile) throws IOException {
```

```

        Properties prop = new Properties();
        FileInputStream fis = new FileInputStream(propertyFile);
        prop.load(fis);
        return prop;
    }
    //根据配置文件的对象来构造 MailContext 对象
    public static MailContext createContext(File propertyFile) throws IOException {
        Properties props = getProperties(propertyFile);
        //如果没有配置 smtp 的端口, 则使用默认的 25 端口
        Integer smtpPort = getInteger(props.getProperty("smtpPort"), 25);
        //如果没有配置 pop3 的端口, 则使用默认的 110 端口
        Integer pop3Port = getInteger(props.getProperty("pop3Port"), 110);
        return new MailContext(null,
                                props.getProperty("account"), props.getProperty("password"),
                                props.getProperty("smtpHost"), smtpPort,
                                props.getProperty("pop3Host"), pop3Port);
    }
    //将参数 s 转换成一个 Integer 对象, 该字符串为空则返回参数中的默认值
    private static Integer getInteger(String s, int defaultValue) {
        if (s == null || s.trim().equals("")) return defaultValue;
        return Integer.parseInt(s);
    }
}

```

`PropertiesUtil` 将用户配置文件中的几个属性读取, 并封装成一个 `MailContext` 对象返回, 以下是 `FileUtil` 的几个常量。

代码清单: `code\foxmail\src\org\crazyit\foxmail\util\FileUtil.java`

```

public class FileUtil {
    //存放所有用户数据的目录
    public static final String DATE_FOLDER = "datas" + File.separator;
    //存放具体某个用户配置的 properties 文件
    public static final String CONFIG_FILE = File.separator + "mail.properties";
}

```

实现以上的代码后, 运行程序, 输入用户名进行登录, 可以看到项目下有一个 `data` 目录, 该目录下就有一个用户目录, 用户目录并且存在于一份 `mail.properties` 的配置文件, 由于现在没有任何的配置, 因此 `mail.properties` 文件的内容为空。这样, 用户登录的功能就已实现, 接下为我们为 `mail.properties` 的添加内容 (用程序添加), 对用户的邮箱进行配置。

## 12.4.2 实现用户信息配置

在 12.2.3 中, 已新建了一个用户的配置界面, 并且描述了如何从主界面进入, 如果用户是第一次进入配置界面, 那么各个配置信息都是空的, 如果不是第一次进入该界面, 那么就需要将以前所配置的信息传递给该界面 (`SetupFrame`), 可以为 `MainFrame` 提供一个 `getMailContext` 的方法, 将对应的 `MailContext` 对象传给 `SetupFrame`, 注意, 当客户端启动的时候, 只能有一个邮箱上下文, 即只能存在一个 `MailContext` 对象。以下是 `SetupFrame` 中初始化各个配置的代码。

代码清单: `code\foxmail\src\org\crazyit\foxmail\ui\SetupFrame.java`

```

//初始化各个配置
this.accountField.setText(ctx.getAccount());
this.passwordField.setText(ctx.getPassword());
this.smtpField.setText(ctx.getSmtpHost());

```

```

this.pop3Field.setText(ctx.getPop3Host());
this.smtpPortField.setText(String.valueOf(ctx.getSmtpPort()));
this.pop3PortField.setText(String.valueOf(ctx.getPop3Port()));

```

**SetupFrame** 在构造的时候,就需要根据 **MainFrame** 传递过来的 **MailContext** 去初始化各个界面组件的值。当用户输入了各个信息后,我们就需要将这些配置信息保存到该用户的配置文件中(即 12.4.1 中的 **mail.properties**)。以下为 **SetupFrame** 部分代码。

代码清单: code\foxmail\src\org\crazyit\foxmail\ui\SetupFrame.java

```

//确定按钮
private void confirm() {
    try {
        //重新设置系统上下文的信息
        MailContext ctx = getMailContext(this.mainFrame.getMailContext());
        //设置已经对信息进行了重新设定
        ctx.setReset(true);
        //将新的上下文写入配置文件中
        PropertiesUtil.store(ctx);
        //设置主界面的 MailContext 对象
        this.mainFrame.setMailContext(ctx);
        //创建存放邮件的目录(并非用户的目录,一个用户可能有多个邮箱地址)
        FileUtil.createFolder(ctx);
        this.setVisible(false);
    } catch (Exception e) {
        JOptionPane.showConfirmDialog(this, e.getMessage(), "警告",
            JOptionPane.OK_CANCEL_OPTION);
    }
}

//根据界面的值封装 MailContext
private MailContext getMailContext(MailContext ctx) {
    //省略从界面得到各个值, 设置 MailContext 的各个属性
    ctx.setAccount(account);
    ctx.setPassword(password);
    ctx.setSmtpHost(smtpHost);
    ctx.setSmtpPort(smtpPort);
    ctx.setPop3Host(pop3Host);
    ctx.setPop3Port(pop3Port);
    //由于重新设置了连接信息, 因此设置 MailContext 的 reset 值为 true
    ctx.setReset(true);
    return ctx;
}

```

用户输入了界面中各个组件的值点击确定后,直接调用以上的 **confirm** 方法, **confirm** 方法调用 **getMailContext** 方法来设置 **MailContext** 的各个属性。在这里需要注意的是,由于重新设置了 **MailContext** 的各个属性,因此需要告诉 **MailContext** 用户更新了配置(设置 **reset** 值为 **true**),那么用户在下次接收邮件的时候(即调用 **getStore** 方法),就可以使用新的配置来重新得到 **Store** 对象。

以上的代码中使用 **PropertiesUtil** 与 **FileUtil** 的两个方法,下面是这两个方法的实现。

代码清单: code\foxmail\src\org\crazyit\foxmail\util\PropertiesUtil.java

```

//保存一个 MailContext 对象, 将它的属性写入文件中
public static void store(MailContext ctx) {
    //得到配置文件 mail.properties
    File propFile = new File(FileUtil.DATE_FOLDER + ctx.getUser() +

```

```

        FileUtil.CONFIG_FILE);
        Properties prop = getProperties(propFile);
        //省略设置各个属性的代码
        //写入配置文件
        FileOutputStream fos = new FileOutputStream(propFile);
        prop.store(fos, "These are mail configs.");
        fos.close();
    }

```

代码清单: code\foxmail\src\org\crazyit\foxmail\util\FileUtil.java

```

//收件箱的目录名
public static final String INBOX = "inbox";
//发件箱的目录名
public static final String OUTBOX = "outbox";
//已发送的目录名
public static final String SENT = "sent";
//草稿箱的目录名
public static final String DRAFT = "draft";
//垃圾箱的目录名
public static final String DELETED = "deleted";
//附件的存放目录名
public static final String FILE = "file";
//创建用户的帐号目录和相关的子目录
public static void createFolder(MailContext ctx) {
    String accountRoot = getAccountRoot(ctx);
    //使用用户当前设置的帐号来生成目录, 例如一个用户叫 user1,
    //有一个邮件账号是 abc@mail.com 那么将会在 datas/user1/下生成一个 abc@mail.com 目录
    mkdir(new File(accountRoot));
    mkdir(new File(accountRoot + INBOX)); //创建 INBOX 目录
    mkdir(new File(accountRoot + OUTBOX)); //发件箱
    mkdir(new File(accountRoot + SENT)); //已发送
    mkdir(new File(accountRoot + DRAFT)); //草稿箱
    mkdir(new File(accountRoot + DELETED)); //垃圾箱
    mkdir(new File(accountRoot + FILE)); //附件存放目录
}
//创建目录的工具方法, 判断目录是否存在
private static void mkdir(File file) {
    if (!file.exists()) file.mkdir();
}
//得到邮件帐号的根目录
private static String getAccountRoot(MailContext ctx) {
    String accountRoot = DATE_FOLDER + ctx.getUser() +
        File.separator + ctx.getAccount() + File.separator;
    return accountRoot;
}

```

PropertiesUtil 的 store 方法主要将 MailContext 的各个属性写到用户的 mail.properties 中, FileUtil 的 createFolder 方法主要用来生成账号的各个目录, 例如有一个用户叫 user1, 那么将会为他在 datas 下面创建一个 user1 的目录, 并且会在 user1 的目录下面再创建五个 Box 目录 (收件箱等)。

SetupFrame 的职责已完成了, 主要用来配置用户信息, 写到配置文件中并生成对应的用户目录, 最后再告诉 MailContext 需要更新配置。下面实现接收邮件的功能。



## 12.5 接收邮件

当我们使用 **JavaMail** 从远程服务器得到邮件后，我们将会得到 **Message** 对象，我们需要对这些 **Message** 对象进行转换，变成我们在 12.3.2 中定义的 **Mail** 对象。而在 12.3.1 中，已经为 **MailContext** 对象定义了返回邮箱仓储对象（**Store**）的方法，因此可以直接使用。

### 12.5.1 接收邮件

新建接口 **MailLoader**，并添加一个接口方法，用于返回 **Mail** 对象的集合，该接口用来接收邮件并对这些邮件进行封装，新建该接口的实现类 **MailLoaderImpl**，具体代码如下。

代码清单：code\foxmail\src\org\crazyit\foxmail\mail\MailLoaderImpl.java

```
public class MailLoaderImpl implements MailLoader {  
    //实现接口方法,返回邮箱中的所有邮件  
    public List<Mail> getMessages(MailContext ctx) {  
        //暂时提供空实现  
        return null;  
    }  
}
```

接下来，我们为 **MailLoaderImpl** 新建一个工具方法，用于获得邮箱中的 **INBOX** 的目录对象，在 **getMessages** 方法中获得所有的邮件。

代码清单：code\foxmail\src\org\crazyit\foxmail\mail\MailLoaderImpl.java

```
//得到邮箱 INBOX  
private Folder getINBOXFolder(MailContext ctx) {  
    Store store = ctx.getStore();  
    try {  
        return store.getFolder("INBOX");  
    } catch (Exception e) {  
        throw new LoadMailException("加载邮箱错误，请检查配置");  
    }  
}  
  
//实现接口方法  
public List<Mail> getMessages(MailContext ctx) {  
    //得到 INBOX 对应的 Folder  
    Folder inbox = getINBOXFolder(ctx);  
    try {  
        //以读写的方式打开 INBOX  
        inbox.open(Folder.READ_WRITE);  
        //得到 INBOX 里的所有信息  
        Message[] messages = inbox.getMessages();  
        for (Message m : messages) {  
            //打印你所需要的信息  
        }  
        inbox.close(false);  
        return null;  
    } catch (Exception e) {  
        throw new LoadMailException(e.getMessage());  
    }  
}
```

以上的代码中的黑体部分，我们可以将得到的各个 **Message** 对象的邮件主题打印出来，可以看到我们需要的结果是否正确，当然，更好的习惯就是编写单元测试进行验证，下面，我们在主界面（**MainFrame**）中使用 **MessageLoader** 的 **getMessages** 方法，验证这些程序是否可以连接上邮件。

代码清单：code\foxmail\src\org\crazyit\foxmail\ui\MainFrame.java

```
//邮箱加载对象
private MailLoader mailLoader = new MailLoaderImpl();
//收取邮件
private Action in = new AbstractAction("收取邮件", new ImageIcon("images/in.gif")) {
    public void actionPerformed(ActionEvent e) {
        receive();
    }
};
//到服务器中收取邮件
public void receive() {
    try {
        List<Mail> newMails = this.mailLoader.getMessages(this.ctx);
    } catch (Exception e) {
        showMessage(e.getMessage());
    }
}
```

那么现在可以启动程序，使用配置界面对邮箱的信息进行配置，再点击收取邮件，可以看到打印的结果，如果不能登录到邮箱或者出现其他异常，那么需要检查邮箱的各个配置是否正确。得到邮箱中的这一系列 **Message** 对象后，我们需要对它们进行封装和排序。首先，提供一些工具方法，用来得到邮件的正文、接收人、发送人、附件、抄送地址、接收日期和邮件大小。

## 12.5.2 获取邮件正文

以下方法返回邮件的正文。

代码清单：code\foxmail\src\org\crazyit\foxmail\mail\MailLoaderImpl.java

```
//返回邮件正文
private StringBuffer getContent(Part part, StringBuffer result) throws Exception {
    if (part.isMimeType("multipart/*")) {
        Multipart p = (Multipart)part.getContent();
        int count = p.getCount();
        //Multipart 的第一部分是 text/plain，第二部分是 text/html 的格式，只需要解析第一部分即可
        if (count > 1) count = 1;
        for(int i = 0; i < count; i++) {
            BodyPart bp = p.getBodyPart(i);
            getContent(bp, result); //递归调用 getContent 方法
        }
    } else if (part.isMimeType("text/*")) {
        result.append(part.getContent()); //遇到文本格式或者 html 格式，直接得到内容
    }
    return result;
}
```

以上代码的黑体部分，递归调用 **getContent** 方法，选择判断参数 **Part** 的类型，如果是 **Multipart**，则再调用 **getContent** 方法获得直接内容，如果是普通的 **text** 类型，则直接加到结果的 **StringBuffer** 对象中。我们可以从 **JavaMail** 的 API 中得知，**Message** 也是 **Part** 类的一个实现类，但 **Message** 是一个抽象类。**Multipart** 与 **Message** 一样，也是 **Part** 的一个实现类，同样也是抽象类。

### 12.5.3 获取邮件收件人

下面的代码和到邮件的收件人。

代码清单：code\foxmail\src\org\crazyit\foxmail\mail\MailLoaderImpl.java

```
//得到一封邮件的所有收件人
private List<String> getAllRecipients(Message m) throws Exception {
    Address[] addresses = m.getAllRecipients();
    return getAddresses(addresses);
}
//工具方法，将参数的地址字符串封装成集合
private List<String> getAddresses(Address[] addresses) {
    List<String> result = new ArrayList<String>();
    if (addresses == null) return result;
    for (Address a : addresses) result.add(a.toString());
    return result;
}
```

得到一个 **Message** 对象，再通过这个对象调用 **getAllRecipients** 方法得到所有收件人的地址，然后再将这些地址封装成一个字符串集合并返回即可。

### 12.5.4 获取邮件的发件人

下面代码得到发件人的地址。

代码清单：code\foxmail\src\org\crazyit\foxmail\mail\MailLoaderImpl.java

```
//得到发送人的地址
private String getSender(Message m) throws Exception {
    Address[] addresses = m.getFrom();
    return MimeUtility.decodeText(addresses[0].toString()); //使用 MimeUtility 进行解码
}
```

我们可以通过 **Message** 的 **getFrom** 方法得到发件人的地址，我们这里只获得一个发件人的地址，当然，如果有必要，也可以得到多个发件人的地址，同样地封装成字符串集合返回，注意，这里还需要使用 **MimeUtility** 的 **decodeText** 方法进行解码。

### 12.5.5 获取邮件的发送日期

下面实现返回邮件日期的方法：

```
//得到接收的日期，优先返回发送日期，其次返回受信日期
private Date getReceivedDate(Message m) throws Exception {
    if (m.getSentDate() != null) return m.getSentDate();
    if (m.getReceivedDate() != null) return m.getReceivedDate();
    return new Date();
}
```

该方法中优先返回邮件的发送日期，我们的客户端程序中只有一个接收日期，你可以为 **Mail** 对象加入多一个发送日期。如果 **Message** 对象的发送日期和接收日期都为空，那么就返回当前的日期。

### 12.5.6 获取邮件的大小

下面实现获得邮件大小的方法。

代码清单: code\foxmail\src\org\crazyit\foxmail\object\Mail.java

```
//将邮件的大小单位定为 k, 并保留两位小数, 参数单位为 byte
public static String getSize(int size) {
    double d = Double.valueOf(size);
    double result = d / 1024;
    return (new java.text.DecimalFormat("#.##")).format(result);
}
```

我们通过 `Message` 的 `getSize` 方法返回的是一个 `int` 类型的数值, 该数值表示该邮件的大小, 单位是字节, 我们需要将这个数值转换为 `k`, 因此上面代码中的 `getSize` 方法用于处理转换的这一过程, 这一过程与邮件相关, 我们可以将这个 `getSize` 方法写到 `Mail` 类中, 作为该类的一个静态方法, 如果不作为静态方法, 那么就需要先创建这个类的实例才可以, 但是我们已经为 `Mail` 对象加入了一个构造器, 邮件的大小就是作为其中一个构造参数。

### 12.5.7 获取邮件的抄送地址

以下方法得到 `Message` 的抄送地址, 并封装成字符串集合。

代码清单: code\foxmail\src\org\crazyit\foxmail\mail\MailLoaderImpl.java

```
//得到抄送的地址
private List<String> getCC(Message m) throws Exception {
    //得到接收人地址,接收人类型是抄送
    Address[] addresses = m.getRecipients(Message.RecipientType.CC);
    return getAddresses(addresses);
}
```

以上代码中的 `getAddresses` 方法, 与 12.5.4 中得到发件人的方法调用的是同一个方法。我们将地址 (`Address`) 对象封装成字符串集合的行为都放到同一个方法中 (`getAddresses`)。

### 12.5.7 获取附件并存放到本地的目录中

下面需要将邮件的附件得到, 并将这些附件放到本地的目录中, 在 12.4.2 中, 我们已经编写了生成本地目录的程序, 生成的本地的目录结构如图 12.5 所示。



图 12.5 用户目录结构

如图 12.5 所示, 用户名的目录是存在于 `datas` 目录下面的, 我们从远程邮件服务器得到邮件对象后, 如果这些对象有附件的话, 就存放到图 12.5 中的 `file` 目录下, 我们这里只需要将附件通过 `Java` 的 `IO` 流放到 `file` 目录下。以下方法得到邮件附件。

代码清单: code\foxmail\src\org\crazyit\foxmail\mail\MailLoaderImpl.java

```
//获得邮件的附件
private List<FileObject> getFiles(MailContext ctx, Message m) throws Exception {
    List<FileObject> files = new ArrayList<FileObject>();
    //是 multipart/mixed 类型, 就进行处理
}
```

```

        if (m.isMimeType("multipart/mixed")) {
            Multipart mp = (Multipart)m.getContent();
            int count = mp.getCount();//得到邮件内容的 Multipart 对象并得到内容中 Part 的数量
            for (int i = 1; i < count; i++) {
                Part part = mp.getBodyPart(i);
                files.add(FileUtil.createFileFromPart(ctx, part)); //在本地创建文件并添加到结果中
            }
        }
        return files;
    }
}

```

注意以上代码的黑体部分，调用了 **FileUtil** 的 **createFileFromPart** 方法，将 **MailContext** 与 **Part** 对象传给 **FileUtil**，让 **FileUtil** 去帮我们得到邮件的附件并存放图 12.5 中的 **file** 目录，以下是 **FileUtil** 的实现。

代码清单：code\foxmail\src\org\crazyit\foxmail\util\FileUtil.java

```

//为附件创建本地文件，目录是登录用户的邮箱名的 file 下
public static FileObject createFileFromPart(MailContext ctx, Part part) {
    String fileRepository = getBoxPath(ctx, FILE); //得到文件存放的目录
    //得到邮件服务器中附件的文件名，并进行解码
    String serverFileName = MimeUtility.decodeText(part.getFileName());
    //生成 UUID 作为在本地系统中唯一的文件标识
    String fileName = UUID.randomUUID().toString();
    //getFileSufix 是得到文件的后缀，这里省略其实现
    File file = new File(fileRepository + fileName + getFileSufix(serverFileName));
    //读写文件
    FileOutputStream fos = new FileOutputStream(file);
    InputStream is = part.getInputStream();
    BufferedOutputStream outs = new BufferedOutputStream(fos);
    //如果附件内容为空 part.getSize 为-1，如果直接 new byte，将抛出异常
    int size = (part.getSize() > 0) ? part.getSize() : 0;
    byte[] b = new byte[size];
    is.read(b);
    outs.write(b);
    //省略 BufferedOutputStream, InputStream 和 FileOutputStream 的 close 方法
    return new FileObject(serverFileName, file);
}

//得到某个目录名字，例如得到 file 的目录, inbox 的目录
public static String getBoxPath(MailContext ctx, String folderName) {
    return getAccountRoot(ctx) + folderName + File.separator;
}
}

```

以上代码中的 **getBoxPath** 方法，是得到图 12.5 中的具体某个目录，例如需要得到 **file** 目录，得可以调用 **getBoxPath**，参数为 **inbox** 即可。以上代码中省略了 **getFileSufix** 方法（黑体部分）的实现，该方法用于得到文件的后缀。因为我们只有一个 **file** 目录用于保存附件，因此可能会发生文件名重复的情况，所以需要使用 **UUID** 重新为文件生成一个文件名，再保存到 **file** 目录，避免了文件名的重复，但是附件的源文件名我们需要保存，所以在最后返回一个 **FileObject** 对象，该对象可以保存文件的源文件名。通过邮件中的 **Part** 对象，可以得到一个 **InputStream** 对象，那么就可以使用 **IO** 流来读写文件，读取服务器上的文件，写到本地的 **file** 目录。

得到 **FileObject** 后，就可以将一封邮件的所有附件封装成一个 **FileObject** 对象的集合，就实现了得到邮件附件的方法。

### 12.5.8 将Message封装为Mail

从 12.5.2 到 12.5.7，我们编写了从 **Message** 中得到邮件的各个信息，那么，我们现在需要将这些信息封装成一个 **Mail** 对象，并添加一个集合中返回。

代码清单：code\foxmail\src\org\crazyit\foxmail\mail\MailLoaderImpl.java

```
//将 javamail 中的 Message 对象转换成本项目中的 Mail 对象
private List<Mail> getMailList(MailContext ctx, Message[] messages) {
    List<Mail> result = new ArrayList<Mail>();
    try {
        //将得到的 Message 对象封装成 Mail 对象
        for (Message m : messages) {
            String xmlName = UUID.randomUUID().toString() + ".xml"; //生成 UUID 的文件名
            String content = getContent(m, new StringBuffer()).toString(); //获得内容
            //得到邮件的各个值
            Mail mail = new Mail(xmlName, getAllRecipients(m), getSender(m),
                m.getSubject(), getReceivedDate(m), Mail.getSize(m.getSize()), hasRead(m),
                content, FileUtil.INBOX);
            mail.setCcs(getCC(m)); //为 mail 对象设置抄送
            mail.setFiles(getFiles(ctx, m)); //设置附件集合
            result.add(mail);
        }
        return result;
    } catch (Exception e) {
        throw new LoadMailException("得到邮件的信息出错: " + e.getMessage());
    }
}
```

以上代码的黑体部分在创建 **Mail** 对象的时候，使用了 12.5.2 到 12.5.7 中所编写的各个工具方法得到邮件的各个信息，并设置到 **Mail** 对象，最后加到集合并返回。这里需要注意的是，**Mail** 中有一个 **xmlName** 的属性，同样需要使用 **UUID** 作为文件名，这是因为我们将会将这个 **Mail** 对象转换成 **XML** 文件，并保存到图 12.5 中的 **inbox** 目录，所以为了保证 **XML** 文件的唯一，就使用 **UUID** 作为 **XML** 的文件名。另外，**Mail** 中有一个 **from** 的属性，该属性表示 **Mail** 对象的来源，像上面的代码一样，这封邮件是通过收件而得来的，因此需要放到 **inbox** 目录。例如，如果在写邮件的时候，将邮件保存到草稿箱，那么这封邮件的 **Mail** 对象来源就是 **draft** 目录（草稿箱）。

### 12.5.9 对邮件进行排序

在 12.5.8 中，我们已经得到了 **Mail** 对象的集合，但是我们还需要帮它们进行排序，以 **Mail** 对象的 **receiveDate** 作为排序依据，将集合进行排序，实现 **Comparator** 接口，再调用 **Collections** 中的 **sort** 方法即可。

代码清单：code\foxmail\src\org\crazyit\foxmail\object\MailComparator.java

```
public class MailComparator implements Comparator {
    //实现 compare 方法
    public int compare(Object o1, Object o2) {
        Mail m1 = (Mail)o1; //将参数强转为 Mail 对象
        Mail m2 = (Mail)o2;
        Date d1 = m1.getReceiveDate(); //获得两个日期进行比较
        Date d2 = m2.getReceiveDate();
        return d2.compareTo(d1);
    }
}
```

```
}
}
```

然后在 MailLoaderImpl 中新建一个 sort 方法，将已经得到的 Mail 对象的集合进行排序。

代码清单：code\foxmail\src\org\crazyit\foxmail\mail\MailLoaderImpl.java

```
//进行时间排序
private void sort(List<Mail> mails) {
    Collections.sort(mails, new MailComparator());
}
```

### 12.5.10 删除邮件服务器上面的邮件

我们从邮件服务器上面得到了所有的邮件后，我们将这些邮件保存到本地的服务器，就可以将邮件服务器中的邮件都删除，保存到本地服务器功能还没有实现，现在可以先实现删除邮件服务器的邮件。为什么需要将远程的邮件删除呢？当然，你也可以选择不用删除，由于我们的程序每次都会去邮件服务器中拿邮件，如果邮件服务回中存在很多的邮件，那么速度将会降低，而我们想得到的只是最新收到的那一份，而以前的那些邮件，都已经保存到了我本地，因此没有必要再收到全部的邮件。为了更好的性能，我们可以将已经收到的邮件从邮件服务器中删除，当有新的邮件进来时，只接收该封新的邮件即可，Foxmail 在不设置的情况下，也是使用这种策略。

下面实现删除邮件服务器的邮件。

代码清单：code\foxmail\src\org\crazyit\foxmail\mail\MailLoaderImpl.java

```
//将邮件数组设置为删除状态
private void deleteFromServer(Message[] messages) throws Exception {
    for (Message m : messages) m.setFlag(Flags.Flag.DELETED, true);
}
```

提供一个 deleteFromServer 方法，设置 Message 的 flag 为 DELETED，封装完 Mail 集合后，就可以调用 deleteFromServer，接下来就可以全部实现 MailLoader 的接口方法 getMessages，在 12.5.1 中我们就已经实现了一部分，现在已经可以全部实现了。

代码清单：code\foxmail\src\org\crazyit\foxmail\mail\MailLoaderImpl.java

```
public List<Mail> getMessages(MailContext ctx) {
    //得到 INBOX 对应的 Folder
    Folder inbox = getINBOXFolder(ctx);
    inbox.open(Folder.READ_WRITE);
    Message[] messages = inbox.getMessages();//得到 INBOX 里的所有信息
    List<Mail> result = getMailList(ctx, messages); //将 Message 数组封装成 Mail 集合
    sort(result); //按照时间降序排序
    //删除邮箱中全部的邮件，那么每次使用邮件系统，只会拿新收到的邮件
    deleteFromServer(messages);
    inbox.close(true); //删除邮件并提交删除状态
    return result;
}
```

代码中的黑体部分，调用了删除邮件的方法设置 Message 对象的 setFlag 方法后，关闭 INBOX 的 Folder 对象时，参数为 true 表示所有 flag 值为 DELETED 的邮件删除，可以理解成提交更改。

### 12.5.11 将Mail对象转换成XML文件并保存到本地目录中

在前面的章节中，我们已经得到了 Mail 对象的集合，那么在收取邮件的时候，需要将这些 Mail 对象保存到本地的目录中，收取邮件，需要保存到 inbox 目录中，具体请看图 12.5 中的目录结构。新建一个接口 SystemHandler，该接口用于处理本地的邮件，包括保存、删除、邮件的移动等操作。添加一

个 `saveInBox` 方法，为这个接口添加实现类，并对接口方法进行实现即可，实现类的代码如下。

代码清单：code\foxmail\src\org\crazyit\foxmail\system\impl\SystemHandlerImpl.java

```
public class SystemHandlerImpl implements SystemHandler {
    public void saveInBox(Mail mail, MailContext ctx) {
        //调用 FileUtil 的方法
        FileUtil.writeToXML(ctx, mail, FileUtil.INBOX);
    }
}
```

在 `SystemHandlerImpl` 中直接调用了 `FileUtil` 的 `writeToXML` 方法，`SystemHandler` 接口是给 `MainFrame` 使用，在前面的章节，通过 `MailLoader` 接口得到 `Mail` 的集合，再使用 `SystemHandler` 接口将这些 `Mail` 对象以 XML 的形式存放到本地的目录中，这一步的操作同样也是放在 `MainFrame` 中进行，那么 `MainFrame` 为什么不直接使用 `FileUtil` 去创建 XML 文件呢？这是由于这样可以使得程序更加清晰，每个接口都有自己固有的职责。以下是 `FileUtil` 的实现。

代码清单：code\foxmail\src\org\crazyit\foxmail\util\FileUtil.java

```
//创建 XStream 对象
private static XStream xstream = new XStream();
//将一个邮件对象使用 XStream 写到 xml 文件中
public static void writeToXML(MailContext ctx, Mail mail, String boxFolder) {
    String xmlName = mail.getXmlName();//得到 mail 对应的 xml 文件的文件名
    String boxPath = getAccountRoot(ctx) + boxFolder + File.separator; //得到对应的目录路径
    File xmlFile = new File(boxPath + xmlName);
    writeToXML(xmlFile, mail);
}
//将一个 mail 对象写到 xmlFile 中
public static void writeToXML(File xmlFile, Mail mail) {
    if (!xmlFile.exists()) xmlFile.createNewFile();
    FileOutputStream fos = new FileOutputStream(xmlFile);
    OutputStreamWriter writer = new OutputStreamWriter(fos, "UTF8");
    //使用 XStream 的 toXML 方法将 Mail 对象转换并输出到 xml 文件
    xstream.toXML(mail, writer);
    writer.close();
    fos.close();
}
```

在 `FileUtil` 中使用了 `XStream`，将 `Mail` 对象转换成对应的 XML 文件，以上代码中，先得到用户的目录，再寻找对应的某个 `Box` 目录（`inbox`、`draft` 等），再将对象写到该目录中的新 XML 文件中，该 XML 文件的文件名使用 `Mail` 对象的 `xmlName` 属性。为主界面对象（`MainFrame`）新建一个工具方法，用于保存邮件对象到本地的 `inbox` 目录。

代码清单：code\foxmail\src\org\crazyit\foxmail\uiMainFrame.java

```
private void saveToInBox(List<Mail> newMails) {
    for (Mail mail : newMails) {
        systemHandler.saveInBox(mail, this.ctx); //生成 xml 来存放这些新的邮件
    }
}
```

那么 `MainFrame` 接收邮件的方法（`receive`）可以写成。

代码清单：code\foxmail\src\org\crazyit\foxmail\uiMainFrame.java

```
List<Mail> newMails = this.mailLoader.getMessage(this.ctx); //得到邮件对象集合
saveToInBox(newMails); //保存到本地的收件箱中
```

保存了邮件到本地的目录后，接下来就可以将这些对象显示到界面的列表中。



### 12.5.12 在界面中显示邮件

得到 Mail 对象的集合后,我们可以将这些对象放到界面中显示。显示 Mail 对象集合的是一个 JTable 对象,我们先给主界面的对象(MainFrame)添加几个集合对象,分别用来代表收件箱、发件箱、已发送、草稿箱和垃圾箱的邮件集合,再使用一个集合来表示当前 JTable 所显示的集合。

代码清单: code\foxmail\src\org\crazyit\foxmail\uiMainFrame.java

```
private List<Mail> inMails; //收件箱的 Mail 对象集合, 代表所有在收件箱中的邮件
private List<Mail> outMails; //发件箱的邮件集合
private List<Mail> sentMails; //成功发送的邮件集合
private List<Mail> draftMails; //草稿箱的邮件集合
private List<Mail> deleteMails; //垃圾箱的邮件集合
private List<Mail> currentMails; //当前界面列表所显示的对象
```

接下来新建一个 MailTableCellRenderer 和 MailListTable 类, MailTableCellRenderer 类用来处理列表的每个单元格显示, MailListTable 继承于 JTable, 表示一个列表对象, 以下为 MailTableCellRenderer 的具体代码。

代码清单: code\foxmail\src\org\crazyit\foxmail\uiMailTableCellRenderer.java

```
public class MailTableCellRenderer extends DefaultTableCellRenderer {
    public Component getTableCellRendererComponent(JTable arg0, Object value,
        boolean arg2, boolean arg3, int arg4, int arg5) {
        //判断该单元格的值是否图片
        if (value instanceof Icon) this.setIcon((Icon)value);
        else this.setText(value.toString());
        return this;
    }
}
```

MailTableCellRenderer 类中重写了 getTableCellRendererComponent 方法, 判断如果该单元格的值是否图片, 再进行设置图片或者文本。MailListTable 的主要作用是设置界面的列表没有表格线和不可编辑, 并且设置只能对列表进行单选, 在这里不贴出具体的代码, 请看本书所附的 MailListTable 类。接下来可以在 MainFrame 中可以这样创建列表对象。

代码清单: code\foxmail\src\org\crazyit\foxmail\uiMainFrame.java

```
public MainFrame(MailContext ctx) {
    //省略其他代码
    this.currentMails = this.inMails; //设置当前显示的邮件集合为收件箱的集合
    DefaultTableModel tableMode = new DefaultTableModel();//邮件列表 JTable
    this.mailListTable = new MailListTable(tableMode);
    //省略 getListColumn 的实现
    tableMode.setDataVector(createViewDatas(this.currentMails), getListColumn());
    setTableFace();//设置邮件列表的样式
    //省略其他代码
}
//没有查看的邮件显示关闭的信封的图片地址
private static String CLOSE_ENVELOP_PATH = "images/envelop-close.gif";
//已经查看的邮件显示打开的信封的图片地址
private static String OPEN_ENVELOP_PATH = "images/envelop-open.gif";
private ImageIcon envelopOpen = new ImageIcon(OPEN_ENVELOP_PATH); //信封打开的 Icon 对象
private ImageIcon envelopClose = new ImageIcon(CLOSE_ENVELOP_PATH); //信封关闭的 Icon 对象
//将邮件数据集合转换成视图的格式
private Vector<Vector> createViewDatas(List<Mail> mails) {
    Vector<Vector> views = new Vector<Vector>();
```

```

        for (Mail mail : mails) {
            Vector view = new Vector();
            //判断邮件对象是否已经被阅读，如果被阅读，则使用打开的信封作为小图标
            if (mail.getHasRead()) view.add(envelopOpen);
            else view.add(envelopClose);
            //省略添加其他字段的代码
        }
        return views;
    }
    //设置邮件列表的样式
    private void setTableFace() {
        //由于打开列需要有信封显示，因此使用 MailTableCellRenderer 作为单元格处理类
        this.mailListTable.getColumn("打开").setCellRenderer(new MailTableCellRenderer());
        //省略其他列的设置代码
    }
}

```

以上代码中创建界面的列表对象，并且将列表的数据设置为当前的 **Mail** 对象集合，那么我们在得到 **Mail** 对象的集合后，就可以刷新一下当前的列表，下面为刷新列表的方法和接收邮件方法的全部实现。

代码清单：code\foxmail\src\org\crazyit\foxmail\ui\MainFrame.java

```

//刷新列表的方法，参数是不同的数据
public void refreshTable() {
    DefaultTableModel tableModel = (DefaultTableModel)this.mailListTable.getModel();
    //使用当前的邮件集合，重新设置列表数据
    tableModel.setDataVector(createViewDatas(this.currentMails), getListColumn());
    setTableFace();//设置列的样式
}
//到服务器中收取邮件
public void receive() {
    try {
        List<Mail> newMails = this.mailLoader.getMessages(this.ctx);
        this.inMails.addAll(0, newMails); //得到 Mail 对象，添加到 inMails 集合中
        saveToInBox(newMails); //保存到本地的收件箱中
        refreshTable();//刷新列表
    } catch (Exception e) {
        showMessage(e.getMessage());
    }
}
}

```

接收邮件的方法已经全部实现了，接下来，可以使用自己的一个邮箱账号进行测试。点击主界面的收件图片，可以看到列表的变化如图 15.6 所示。

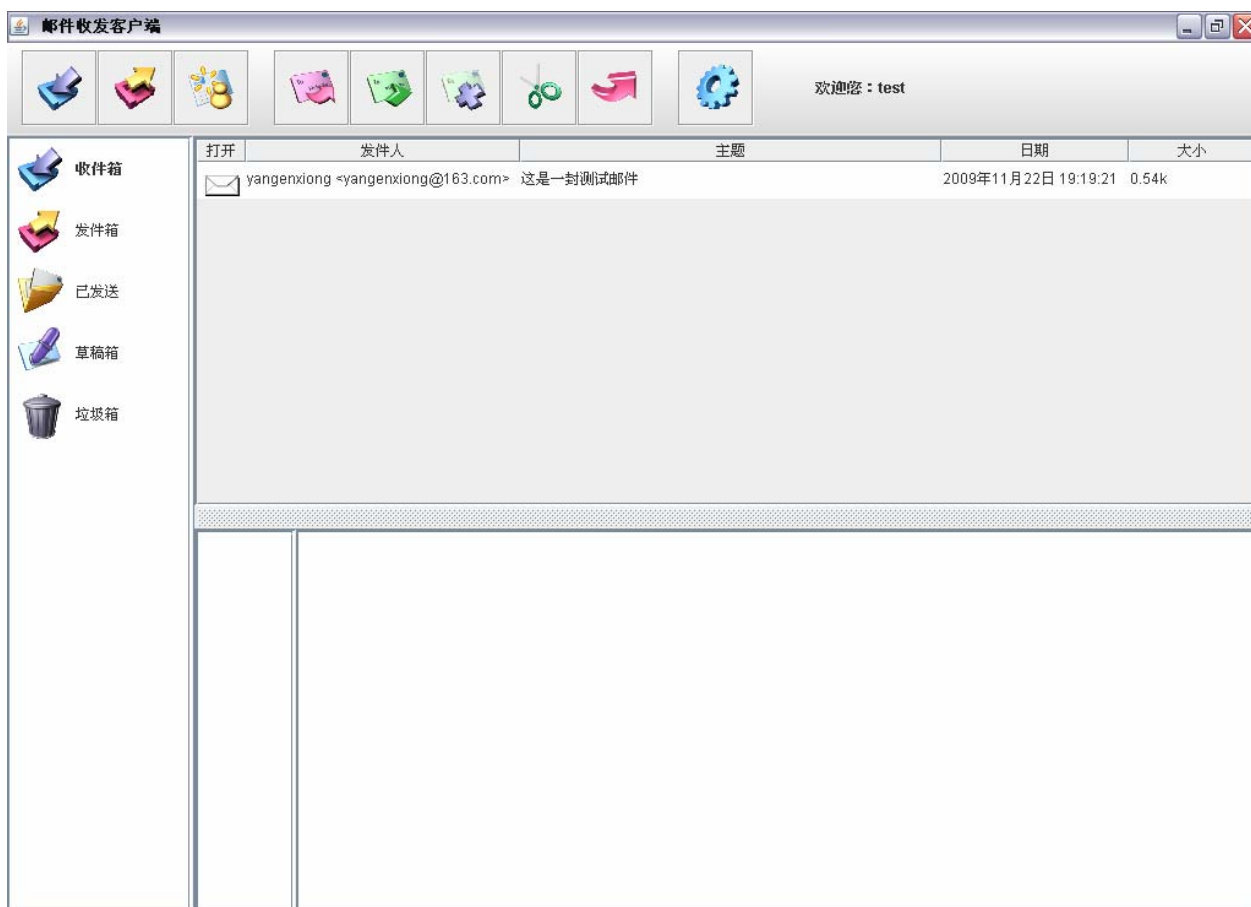


图 12.6 接收邮件的效果

当点击一次收取邮件时，客户端将会从远程邮件服务器接收邮件，并转换成列表的数据格式展现到列表中。在图 12.6 中我们可以看到，“打开”列的信封图标是关闭的，下面章节将会实现将这个 **Mail** 对象的打开操作，打开一封邮件时，信封的图标需要变成打开，并且邮件的内容需要显示到具体的区域中。

### 12.5.13 编写任务调度器接收邮件

很多时候，用户并不需要手动去接收邮件，我们可以提供一个任务调度器，隔一段时间让客户端去邮件服务器下载邮件，新建一个任务调度器，并在主界面初始化的时候运行。

代码清单：code\foxmail\src\org\crazyit\foxmail\uiMainFrame.java

```
public class MainFrame extends JFrame {
    public MainFrame(MailContext ctx) {
        //省略其他代码
        Timer timer = new Timer();
        timer.schedule(new ReceiveTask(this), 10000, this.receiveInterval);
    }
    //省略其他代码
}

class ReceiveTask extends TimerTask {
    private MainFrame mainFrame;
    public ReceiveTask(MainFrame mainFrame) {
```

```

        this.mainFrame = mainFrame;
    }
    public void run() {
        this.mainFrame.getMailContext().getStore();
        //调用主界面的 receive 方法
        this.mainFrame.receive();
    }
}

```

在以上的代码中，我们设置每隔 10000 毫秒（10 秒）收取邮件一次，那么用户就不需要手动的去收取邮件。

## 12.6 初始化界面数据

在 12.5 中，我们编写了从邮件服务器接收邮件，转换成 Mail 对象，并在本地的目录中保存对应的目录中，如果用户并不是第一次登录，系统的目录中已经存在该用户的相关配置及邮件（XML 文件），那么我们需要将这些 XML 文件转换成系统的 Mail 对象，再显示在对应的列表中。

### 12.6.1 转换XML文件为Mail对象

当用户登录时，我们可以得到用户所有的配置信息，并且可以在 MailContext 对象中体现出来，那么我们可以根据 MailContext 对象去加载用户邮箱（本地目录）中的所有邮件，再显示到界面。在前面的章节中提到，一个用户的具体目录如图 12.5 所示，用户名为根目录，下面是这个用户的各个邮箱账号目录，每一个邮箱目录下面有多个目录，分别代码收件箱、发件箱、已发送、草稿箱和垃圾箱，还有附件存放的目录。

为 FileUtil 添加查找文件的方法，将几个 Box 目录下的所有 XML 文件都读取出来，返回文件集合。  
代码清单：code\foxmail\src\org\crazyit\foxmail\util\FileUtil.java

```

//从相应的 box 中得到全部的 xml 文件
public static List<File> getXMLFiles(MailContext ctx, String box) {
    String rootPath = getAccountRoot(ctx);
    String boxPath = rootPath + box;
    File boxFolder = new File(boxPath); //得到某个 box 的目录
    List<File> files = filterFiles(boxFolder, ".xml"); //对文件进行后缀过滤
    return files;
}
//从一个文件目录中，以参数文件后缀 suffix 为条件，过滤文件
private static List<File> filterFiles(File folder, String suffix) {
    List<File> result = new ArrayList<File>();
    File[] files = folder.listFiles();
    if (files == null) return new ArrayList<File>();
    for (File f : files) {
        if (f.getName().endsWith(suffix)) result.add(f);
    }
    return result;
}

```

当调用 getXMLFiles 方法后，就得到具体某个 Box 目录下面的所有 XML，得到这些 XML 文件后，再次使用 XStream 进行转换，将 XML 文件转换成 Mail 对象。新建一个 SystemLoader 的接口，专门用于读取本地系统中的 Mail 对象，为该接口添加接口方法并提供实现类 SystemLoaderImpl。

代码清单: code\foxmail\src\org\crazyit\foxmail\mail\SystemLoaderImpl.java

```
//实现接口方法
public List<Mail> getInBoxMails(MailContext ctx) {
    //先从与用户对应的 inbox 中得到全部的 xml 文件
    List<File> xmlFiles = FileUtil.getXMLFiles(ctx, FileUtil.INBOX);
    //将 XML 文件转换成 Mail 对象
    List<Mail> result = convert(xmlFiles, ctx);
    return result;
}
//将 xml 文件转换成 Mail 对象, 并排序
private List<Mail> convert(List<File> xmlFiles, MailContext ctx) {
    List<Mail> result = new ArrayList<Mail>();
    for (File file : xmlFiles) {
        Mail mail = FileUtil.fromXML(ctx, file); //将 xml 转换成 Mail 对象
        result.add(mail);
    }
    sort(result);
    return result;
}
//按照时间降序排序
private void sort(List<Mail> mails) {
    Collections.sort(mails, new MailComparator());
}
```

以上代码中, 得到 `File` 对象的集合后, 再调用 `convert` 方法进行转换, 其中 `convert` 又调用了 `FileUtil` 中的 `fromXML` 文件进行转换, 得到最终的 `Mail` 集合后, 调用 `sort` 方法进行时间排序, `sort` 方法中的 `MailComparator` 已经在 12.5.9 中实现。以下是 `FileUtil` 的 `fromXML` 的方法实现。

代码清单: code\foxmail\src\org\crazyit\foxmail\util\FileUtil.java

```
//将一份 xml 文档转换成 Mail 对象
public static Mail fromXML(MailContext ctx, File xmlFile) {
    try {
        FileInputStream fis = new FileInputStream(xmlFile);
        //调用 XStream 的转换方法将文件转换成对象
        Mail mail = (Mail)xstream.fromXML(fis);
        fis.close();
        return mail;
    } catch (Exception e) {
        throw new FileNotFoundException("转换数据异常: " + xmlFile.getAbsolutePath());
    }
}
```

`XStream` 为我们提供了十分方便的转换方法, 直接调用一个方法就可以将文件转换成数据, 当然, `XStream` 的功能远不止于此, 由于本章的主要内容是邮件, 因此 `XStream` 更深入的使用不再详细描述。

当我们得到 `inbox` 目录的邮件对象时, 就可以将这些对象显示到界面, 除了收件箱 `inbox` 目录的邮件 (XML 文件) 外, 还需要有其他的邮件, 包括发件箱、已发送、草稿箱和垃圾箱。下面继续实现这几个 `Box` 的邮件读取, 为 `SystemLoader` 接口新增方法并提供实现。

代码清单: code\foxmail\src\org\crazyit\foxmail\mail\SystemLoaderImpl.java

```
//实现接口方法, 得到垃圾箱的邮件
public List<Mail> getDeletedBoxMails(MailContext ctx) {
    return getMails(ctx, FileUtil.DELETED);
}
```

```

//实现接口方法, 得到草稿箱的邮件
public List<Mail> getDraftBoxMails(MailContext ctx) {
    return getMails(ctx, FileUtil.DRAFT);
}
//实现接口方法, 得到发件箱中的邮件
public List<Mail> getOutBoxMails(MailContext ctx) {
    return getMails(ctx, FileUtil.OUTBOX);
}
//实现接口方法, 得到已发送的邮件
public List<Mail> getSentBoxMails(MailContext ctx) {
    return getMails(ctx, FileUtil.SENT);
}
//工具方法
private List<Mail> getMails(MailContext ctx, String box) {
    List<File> xmlFiles = FileUtil.getXMLFiles(ctx, box);
    List<Mail> result = convert(xmlFiles, ctx);
    return result;
}

```

以上的各个 **Box** 的邮件读取代码与前面收件箱的读取代码无异, 而且显得更加简洁, 都是到对应的目录中读取 XML 文件, 再使用 **XStream** 进行转换。

## 12.6.2 在各个Box间切换

在各个 **Box** 间切换, 说白了就是点击导航树, 再进行界面数据的切换, 我们的主界面列表中只有一个 **JTable** 对象, 当用户点击了导航树的某个节点后, 我们需要将不同的数据设置到 **JTable** 中。在 12.6.1 中得到各个 **Box** 的 **Mail** 对象后, 在主界面 (**MailFrame**) 中就可以将这些 **Mail** 对象传递给界面中对应的集合。

代码清单: code\foxmail\src\org\crazyit\foxmail\ui\MainFrame.java

```

//MainFrame 构造器
public MainFrame(MailContext ctx) {
    this.ctx = ctx;
    //初始化各个列表集合
    initMails();
    //设置当前显示的邮件集合为收件箱的集合
    this.currentMails = this.inMails;
    //省略其他代码
}
//初始化时创建各个 box 中的数据
private void initMails() {
    this.inMails = this.systemLoader.getInBoxMails(this.ctx);
    this.draftMails = this.systemLoader.getDraftBoxMails(this.ctx);
    this.deleteMails = this.systemLoader.getDeletedBoxMails(this.ctx);
    this.outMails = this.systemLoader.getOutBoxMails(this.ctx);
    this.sentMails = this.systemLoader.getSentBoxMails(this.ctx);
}

```

在 **MainFrame** 构造器中初始化各个邮件集合, 用户第一次进来客户端的时候, 就可以显示收件箱的邮件 (设置当前显示的邮件集合)。那么, 用户点了导航树中的某个节点时, 就可以将当前显示的邮件集合转换成对应的邮件集合, 再刷新列表, 就可以达到数据的切换的效果。在 12.2.2 中, 我们创建了主界面的对象, 并为其新建了导航树的鼠标监听类 **SailTreeListener**, 当时该类的 **mousePressed** 方法

我们提供了空实现，现在可以对其进行实现，先为 `MainFrame` 加入一个 `select` 方法，用于处理导航树的点击。

代码清单：code\foxmail\src\org\crazyit\foxmail\uiMainFrame.java

```
//当鼠标点击了导航树的某个节点时，触发这个方法
public void select() {
    MailBox box = getSelectBox();
    //判断 MailBox 的类，并显示不同的数据
    if (box instanceof InBox) {
        this.currentMails = this.inMails;
    } else if (box instanceof OutBox) {
        this.currentMails = this.outMails;
    } else if (box instanceof SentBox) {
        this.currentMails = this.sentMails;
    } else if (box instanceof DraftBox) {
        this.currentMails = this.draftMails;
    } else {
        this.currentMails = this.deleteMails;
    }
    //刷新列表
    refreshTable();
}
//获得当前选中的 box
private MailBox getSelectBox() {
    TreePath treePath = this.tree.getSelectionPath();
    if (treePath == null) return null;
    //获得选中的 TreeNode
    DefaultMutableTreeNode node = (DefaultMutableTreeNode)treePath.getLastPathComponent();
    return (MailBox)node.getUserObject();
}
```

以上的 `select` 方法，在导航树中得到某个 `MailBox` 对象，由于在创建树的时候，我们是使用 `MailBox` 的实现类去创建节点的（具体请看 12.2.2），因此当点击了树上的某个节点时，就可以得到相应的 `MailBox` 对象，再根据这些 `MailBox` 的实现类进行判断，显示相应的数据。接下来去实现 `SailTreeListener` 监听器：

```
public void mousePressed(MouseEvent e) {
    this.mainFrame.select();
}
```

运行程序并对导航树进行点击，即可看到效果。

## 12.7 操作邮件

这里所讲的操作邮件包括查看邮件、删除邮件、彻底删除邮件和还原邮件。当用户选择了某一份邮件查看时，该邮件的信息将会显示在具体信息的区域，可以显示邮件的主题、正文、收件时间、发送人和附件等信息。用户点击了删除邮件时，就需要将该邮件放到垃圾箱（`deleted` 目录）中，只需要将邮件对应的 `XML` 文件放到 `deleted` 目录。彻底删除邮件，就需要将邮件从本地的目录中删除，需要删除 `XML` 文件，还有该邮件所对应的所有附件，一旦彻底删除，将不可被还原。还原邮件只适合于操作垃圾箱中的邮件，在 `Mail` 对象中，我们已经有一个 `from` 属性（具体请看 12.3.2），表明这个邮件对象的来源，那么点击还原时，就可以还原到相应的目录中。

### 12.7.1 查看邮件

点击了列表的某行邮件记录，就可以将该邮件的信息显示到相应的区域，在列表中的每一行数据，都对应一个 Mail 对象，并且这个 Mail 对象的 xmlName 属性是唯一的，我们就可以根据这个属性去找到这个 Mail 对象。以下代码实现查看邮件。

代码清单：code\foxmail\src\org\crazyit\foxmail\ui\MainFrame.java

```
//当读取邮件后将图片改变为打开信封图片
private void openEnvelop() {
    int row = this.mailListTable.getSelectedRow();
    int column = this.mailListTable.getColumn("打开").getModelIndex();
    this.mailListTable.setValueAt(this.envelopOpen, row, column);
    //重新保存邮件状态到 xml 文件
    this.systemHandler.saveMail(this.currentMail, this.ctx);
}
//获取在列表中所选择的 Mail 对象
private Mail getSelectMail() {
    String xmlName = getSelectXmlName();
    return getMail(xmlName, this.currentMails);
}
//从集合中找到 xmlName 与参数一致的 Mail 对象
private Mail getMail(String xmlName, List<Mail> mails) {
    for (Mail m : mails) {
        if (m.getXmlName().equals(xmlName))return m;
    }
    return null;
}
//查看一封邮件
private void viewMail() {
    this.mailTextArea.setText("");
    Mail mail = getSelectMail();
    this.mailTextArea.append("发送人:   " + mail.getSender());
    this.mailTextArea.append("\n");
    this.mailTextArea.append("抄送:   " + mail.getCCString());
    this.mailTextArea.append("\n");
    this.mailTextArea.append("收件人:   " + mail.getReceiverString());
    this.mailTextArea.append("\n");
    this.mailTextArea.append("主题:   " + mail.getSubject());
    this.mailTextArea.append("\n");
    this.mailTextArea.append("接收日期:   " + dateFormat.format(mail.getReceiveDate()));
    this.mailTextArea.append("\n\n");
    this.mailTextArea.append("邮件正文:   ");
    this.mailTextArea.append("\n\n");
    this.mailTextArea.append(mail.getContent());
    this.fileList.setListData(mail.getFiles().toArray());//添加附件
    this.currentMail = mail;//设置当前被打开的邮件对象
    //如果邮件没有被查看过，就修改图标，并保存已经打开的状态
    if (!mail.getHasRead()) {
        mail.setHasRead(true);//设置邮件已经被查看
        openEnvelop();//更新信封图标
    }
}
```



```
    }  
}
```

查看邮件，需要得到选中行对应数据的 `xmlName` 列的值，再从当前显示的列表中查找到 `Mail` 对象，得到该对象后，再向显示区域添加文本和添加附件，最后设置该邮件的状态为已经被打开。以上代码的黑体部分使用了 `SystemHandler` 的 `saveMail` 方法，由于显示的 `Mail` 对象的 `hasRead` 值已从 `false` 变为 `true`，那么就要相应的去修改 XML 文件，并不需要找到对应的节点去修改，只需要将 `Mail` 对象重新生成一次 XML 文件即可，以下是 `SystemHandler` 中的 `saveMail` 方法的实现。

代码清单：code\foxmail\src\org\crazyit\foxmail\system\impl\SystemHandlerImpl.java

```
public void saveMail(Mail mail, MailContext ctx) {  
    //需要寻找该 Mail 对象所对应的 xml 文件，根据 id 去找文件  
    File xmlFile = getMailXmlFile(mail.getXmlName(), ctx);  
    //找到该份 XML 文件后，再将 Mail 对象重新转换成 XML 并写入  
    FileUtil.writeToXML(xmlFile, mail);  
}  
//从所有的邮件中查找名字为 xmlName 的 xml 文件  
private File getMailXmlFile(String xmlName, MailContext ctx) {  
    //从所有的 Box 目录中得到全部的 XML 文件，再返回对应的 XML 文件  
    List<File> allXMLFiles = getAllFiles(ctx);  
    for (File f : allXMLFiles) {  
        if (f.getName().equals(xmlName)) return f;  
    }  
    return null;  
}
```

以上代码的黑体部分，得到全部的 XML 文件，该方法只是将所有 `Box` 目录中的 XML 文件加到一个集合中，再对这个文件集合进行遍历，找出符合条件的 XML 文件。一旦邮件被打开，那么它的状态将永远是已读，运行程序可以看到效果如图 12.7 所示。

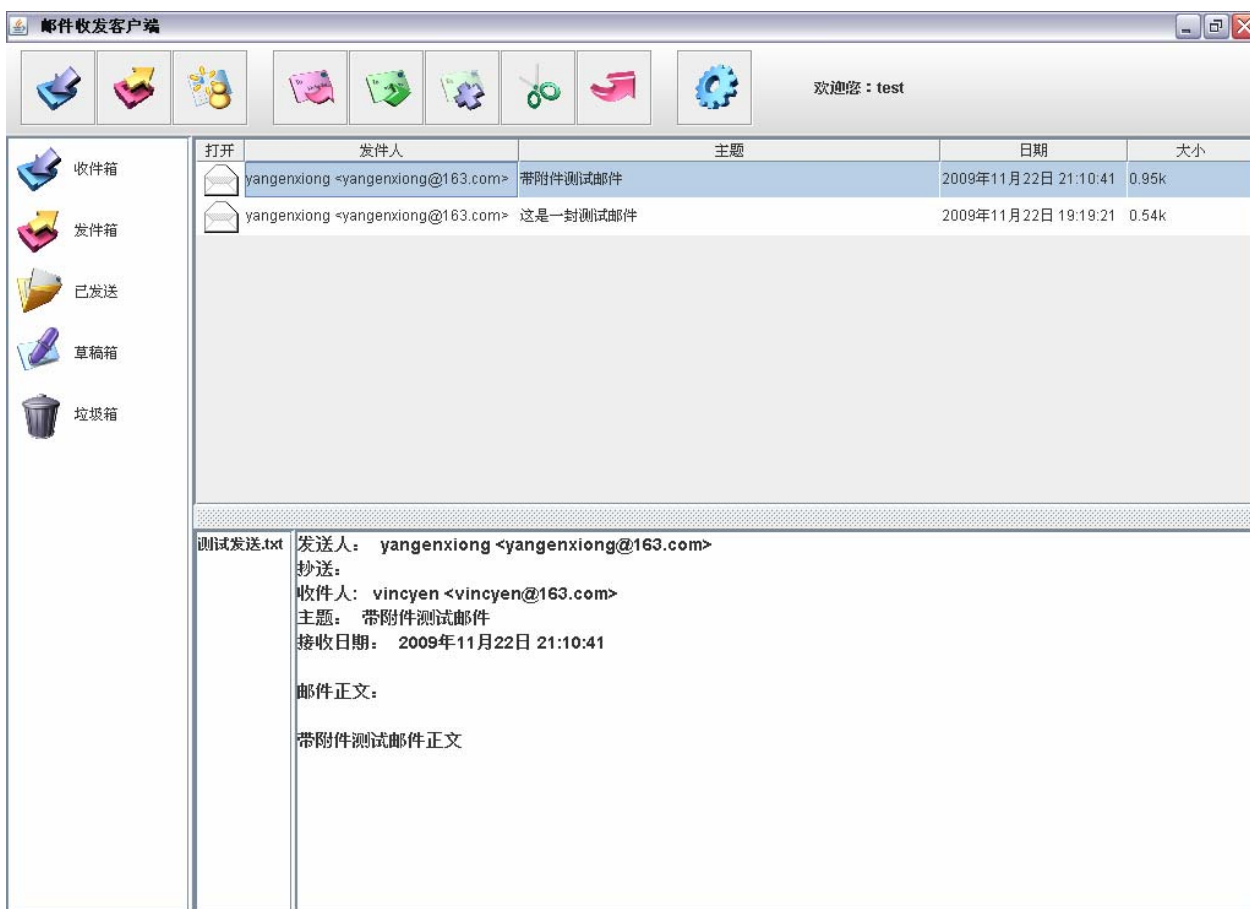


图 12.7 查看邮件

### 12.7.2 查看附件

在 12.7.1 中，已经实现了查看邮件的功能，查看邮件时，邮件的附件会显示到显示区的列表中，那么，我们需要对附件进行操作时，就可以双击某份附件，再选择具体的操作。我们在这里给用户定义了两个操作，一个是直接打开附件，另外一个是将附件另存为。如果直接打开附件，可以使用 `cmd` 命令直接打开，如果是另存为操作，也可以使用 `cmd` 的 `copy` 命令进行复制，如果是 Linux 操作系统，可以使用 `cp` 命令。

下面，先实现文件的 `copy`，在 `FileUtil` 中新建一个 `copy` 的方法。

代码清单：code\foxmail\src\org\crazyit\foxmail\util\FileUtil.java

```
//复制文件的方法
public static void copy(File sourceFile, File targetFile) {
    //调用 copy 命令
    Process process = Runtime.getRuntime().exec("cmd /c copy \"" +
        sourceFile.getAbsolutePath() + "\" \"" +
        targetFile.getAbsolutePath() + "\"");
    //等待命令执行完成
    process.waitFor();
}
```

准备了 `copy` 方法后，就可以去实现附件的操作，需要为存放附件的列表添加监听器，附件列表在主界面（`MainFrame`）中是一个 `JList` 对象，我们新建一个鼠标事件监听器即可。

代码清单: code\foxmail\src\org\crazyit\foxmail\ui\MainListMouseListener.java

```
//主界面中邮件附件列表的监听器
public class MainListMouseListener extends MouseAdapter {
    public void mouseClicked(MouseEvent e) {
        if (e.getClickCount() == 2) {
            JList list = (JList)e.getSource();
            FileObject file = (FileObject)list.getSelectedValue();
            if (file == null) return;
            handle(file);
        }
    }
    //处理方法, 可以给子类去调用
    public void handle(FileObject file) {
        int result = JOptionPane.showOptionDialog(null, "请选择操作", "选择",
            0, JOptionPane.QUESTION_MESSAGE, null,
            new Object[]{"打开", "另存为", "取消"}, null);
        if (result == 0) {
            openFile(file); //打开操作
        } else if (result == 1) {
            saveAs(file); //另存为操作
        }
    }
    //打开操作
    public void openFile(FileObject file) {
        Runtime.getRuntime().exec("cmd /c \"" + file.getFile().getAbsolutePath() + "\"");
    }
    //另存为操作, 打开文件选择器
    public void saveAs(FileObject file) {
        FolderChooser chooser = new FolderChooser(file);
        chooser.showSaveDialog(null);
    }
}
//文件选择器类
class FolderChooser extends JFileChooser {
    //需要另存为的文件
    private FileObject sourceFile;
    public FolderChooser(FileObject sourceFile) {
        this.sourceFile = sourceFile;
        //只能选目录
        this.setFileSelectionMode(DIRECTORIES_ONLY);
    }
    //在文件选择器中选择了文件或者目录后
    public void approveSelection() {
        File targetFile = getSelectedFile();
        if (targetFile.isDirectory()) {
            //如果用户选的是目录, 即没有输入新的文件名, 则用 sourceName 作为文件名
            File newFile = new File(targetFile.getAbsolutePath() + File.separator
                + this.sourceFile.getSourceName());
            FileUtil.copy(this.sourceFile.getFile(), newFile);
        } else {
```

```

        //用户输入了新的文件名, 直接复制
        FileUtil.copy(this.sourceFile.getFile(), targetFile);
    }
    super.approveSelection();
}
}

```

当用户在附件列表双击了某份附件时, 就弹出提示, 询问具体的操作, 是打开还是另存为, 如果打开, 执行以上代码的 `openFile` 方法, 由于邮件已经下载到了本地, 这些附件是存在于账号目录的 `file` 目录下, 因此可以直接打开。如果选择了另存为, 那么就显示文件选择器, 让用户选择该文件的存放目录, 再调用 `FileUtil` 的 `copy` 方法, 存在于 `file` 目录中的附件复制到用户选择的目录中。

注意: 在另存为的时候, 需要重新生成一份文件, 文件名是 `FileObject` 的 `sourceName` 属性。查看附件的效果如图 12.8 所示。



图 12.8 查看附件

### 12.7.3 删除邮件

删除邮件, 只是简单的将该邮件对应的 XML 文件放到 `deleted` 目录即可。我们需要为 `SystemHandler` 接口添加一个 `delete` 方法, 用于删除邮件。具体的代码如下。

代码清单: `code\foxmail\src\org\crazyit\foxmail\system\impl\SystemHandlerImpl.java`

```

//实现接口的 delete 方法
public void delete(Mail mail, MailContext ctx) {
    //找到对应的 xml 文件
    File file = getMailXmlFile(mail.getXmlName(), ctx);
    //删除文件, 并在 deleted 目录中创建新的文件
    file.delete();
    //创建新的 xml 文件
    FileUtil.writeToXML(ctx, mail, FileUtil.DELETED);
}

```

删除方法比较简单, 只要有 `Mail` 对象, 就可以得到对应的 XML 文件, 再对其进行删除, 再以 `Mail` 对象为基础, 重新生成一份新的 XML 文件到 `deleted` 目录下。以上代码中的黑体部分的 `getMailXmlFile` 方法已经在 12.7.1 中实现。以下是主界面 (`MainFrame`) 的删除邮件代码。

代码清单: `code\foxmail\src\org\crazyit\foxmail\uiMainFrame.java`

```

//删除邮件
public void delete() {
    //getSelectMail 已经实现
    Mail mail = getSelectMail();
    //判断垃圾箱中是否有该份邮件(已经被放到垃圾箱中), 有的话不进行处理
    if (!this.deleteMails.contains(mail)) {
        this.currentMails.remove(mail); //从当前的集合中删除
        this.deleteMails.add(0, mail); //加到垃圾箱的集合中
        //将邮件对应的 xml 文件放到 deleted 的目录中
    }
}

```

```

        this.systemHandler.delete(mail, this.ctx);
    }
    this.currentMail = null;
    refreshTable();//刷新列表
}

```

### 12.7.4 彻底删除邮件

彻底删除邮件与删除邮件一样，而且更加简单，只是将邮件对应的 XML 文件删除，再删除该邮件所对应的附件即可。在 **SystemHanlder** 中新增一个接口方法。

代码清单：code\foxmail\src\org\crazyit\foxmail\system\impl\SystemHandlerImpl.java

```

//实现接口方法
public void realDelete(Mail mail, MailContext ctx) {
    //找到对应的 xml 文件
    File xmlFile = getMailXmlFile(mail.getXmlName(), ctx);
    //得到所有的附件并删除
    List<FileObject> files = mail.getFiles();
    //删除附件
    for (FileObject f : files) f.getFile().delete();
    //删除 xml 文件
    if (xmlFile.exists()) xmlFile.delete();
}

```

只要点击了彻底删除邮件，就会从本地的系统中彻底删除该邮件的 XML 文件与附件，当删除完目录中的文件后，还需要更新列表，以下是主界面（**MainFrame**）的彻底删除代码。

代码清单：code\foxmail\src\org\crazyit\foxmail\uiMainFrame.java

```

//彻底删除一封邮件
private void realDelete() {
    Mail mail = getSelectMail();
    //从当前显示的集合中删除
    this.currentMails.remove(mail);
    //删除 xml 文件和对应的附件
    this.systemHandler.realDelete(mail, this.ctx);
    this.currentMail = null;
    refreshTable();//刷新列表
}

```

### 12.7.5 还原邮件

当用户删除了一份邮件后，需要对其进行还原，可以使用这个功能，我们在设计 **Mail** 对象的时候，已经为这个对象加入了一个 **from** 的属性，表示这个 **Mail** 对象的出处，如果是从远程的邮件服务器接收的，那么这个值就是收件箱的目录名称，如果是写邮件时加到草稿箱的，那么这个值就是草稿箱的目录名称，只需要得到 **Mail** 对象，就可以对其进行还原，为 **SystemHandler** 接口添加方法并对其进行实现。

代码清单：code\foxmail\src\org\crazyit\foxmail\system\impl\SystemHandlerImpl.java

```

//实现接口方法
public void revert(Mail mail, MailContext ctx) {
    //找到对应的 xml 文件
    File xmlFile = getMailXmlFile(mail.getXmlName(), ctx);
    //删除该文件，再还原到原来的目录中
}

```

```
xmlFile.delete();
FileUtil.writeToXML(ctx, mail, mail.getFrom());
}
```

该实现比较简单，与删除邮件一样，将存在于 `deleted` 目录中的 XML 文件删除，再重新生成一份 XML 文件到邮件原来存在的目录即可，以下是 `MainFrame` 对还原邮件的实现。

代码清单：`code\foxmail\src\org\crazyit\foxmail\ui\MainFrame.java`

```
//还原 Mail 对象到各个相应的集合
private void revertMailToList(Mail mail) {
    if (mail.getFrom().equals(FileUtil.INBOX)) {
        this.inMails.add(mail);
    } else if (mail.getFrom().equals(FileUtil.SENT)) {
        this.sentMails.add(mail);
    } else if (mail.getFrom().equals(FileUtil.DRAFT)) {
        this.draftMails.add(mail);
    } else if (mail.getFrom().equals(FileUtil.OUTBOX)) {
        this.outMails.add(mail);
    }
}

//还原邮件
public void revert() {
    Mail mail = getSelectMail();
    //垃圾箱包含这个 Mail 对象才进行还原
    if (this.deleteMails.contains(mail)) {
        //从垃圾箱集合中删除
        this.deleteMails.remove(mail);
        //操作文件，并 deleted 目录中的 xml 中
        this.systemHandler.revert(mail, this.ctx);
        //还原到各个集合中
        revertMailToList(mail);
    }
    this.currentMail = null;
    refreshTable();//刷新列表
}
```

`MainFrame` 中实现还原邮件，得到需要删除的邮件对象后，再对其进行判断，看下该对象是否已经在垃圾箱中，如果不是，那么就不进行任何操作。还原邮件时先从垃圾箱对应的邮件集合中删除，再调用 `SystemHandler` 的方法还原 XML 文件，最后再还原到 `MainFrame` 中相应的集合。

注意：还原到 `MainFrame` 中相应的集合时，需要对该邮件的 `from` 属性进行判断，才知道需要还原到哪个集合中。

## 12.8 发送邮件

前面的章节已经实现了三个界面的功能，剩下的只有编写邮件的界面没有完成，实现了发送邮件的功能后，我们就可以再回过头去实现主界面的回复功能、转发功能和发送功能，这些功能与编写邮件的功能大致类似，如果是回复功能的话，只需要将原邮件的发件人作为收件人来发待即可。

### 12.8.1 实现发送邮件的功能

发送邮件，我们需要使用 **JavaMail** 的方法进行发送，发送成功，还需要将发送的 **Mail** 对象保存到已发送的 **Box** 目录（**sent**），如果发送失败，需要保存到发件箱中，让用户在下次再继续发送。发送邮件我们使用的是 **SMTP** 的协议，与接收邮件时创建的 **Store** 对象一样，我们在 **MailContext** 对象中新建一个发送邮件的 **Session** 属性，并提供返回该属性的方法：

代码清单：code\foxmail\src\org\crazyit\foxmail\ui\MailContext.java

```
private Session session;
//返回 Session 对象
public Session getSession() {
    //重置了信息，设置 session 为 null
    if (this.reset) {
        this.session = null;
        this.store = null;
        this.reset = false;
    }
    if (this.session == null) {
        Properties props = System.getProperties();
        //设置 SMTP 服务器和端口
        props.put("mail.smtp.host", this.getSmtHost());
        props.put("mail.smtp.port", this.getSmtPort());
        props.put("mail.smtp.auth", true);
        //创建 Session 对象
        Session sendMailSession = Session.getDefaultInstance(props, getAuthenticator());
        this.session = sendMailSession;
    }
    return this.session;
}
```

**Session** 对象的获得，与接收邮件的 **Store** 对象获得一样，当为空的时候再创建，如果用户重新配置了邮箱的信息，那么就设置 **Session** 对象为空并再重新创建，这样做可以增加性能。

新建一个发送邮件的接口 **MailSender**，添加一个接口方法 **send** 并为其提供实现。

代码清单：code\foxmail\src\org\crazyit\foxmail\mail\MailSenderImpl.java

```
//实现接口方法
public Mail send(Mail mail, MailContext ctx) {
    try {
        Session session = ctx.getSession();
        Message mailMessage = new MimeMessage(session);
        //设置发件人地址
        Address from = new InternetAddress(ctx.getUser() + "<" + ctx.getAccount() + ">");
        mailMessage.setFrom(from);
        //设置所有收件人的地址
        Address[] to = getAddress(mail.getReceivers());
        mailMessage.setRecipients(Message.RecipientType.TO, to);
        //设置抄送人地址
        Address[] cc = getAddress(mail.getCcs());
        mailMessage.setRecipients(Message.RecipientType.CC, cc);
        //设置主题
        mailMessage.setSubject(mail.getSubject());
        //发送日期
```

```

        mailMessage.setSentDate(new Date());
        //构建整封邮件的容器
        Multipart main = new MimeMultipart();
        //正文的 body
        BodyPart body = new MimeBodyPart();
        body.setContent(mail.getContent(), "text/html; charset=utf-8");
        main.addBodyPart(body);
        //处理附件
        for (FileObject f : mail.getFiles()) {
            //每个附件的 body
            MimeBodyPart fileBody = new MimeBodyPart();
            fileBody.attachFile(f.getFile());
            //为文件名进行转码
            fileBody.setFileName(MimeUtility.encodeText(f.getSourceName()));
            main.addBodyPart(fileBody);
        }
        //将正文的 Multipart 对象设入 Message 中
        mailMessage.setContent(main);
        Transport.send(mailMessage);
        return mail;
    } catch (Exception e) {
        throw new SendMailException("发送邮件错误, 请检查邮箱配置及邮件的相关信息");
    }
}
//将字符串集合转换成一个 Address 的数组
private Address[] getAddress(List<String> addList) throws Exception {
    Address[] result = new Address[addList.size()];
    for (int i = 0; i < addList.size(); i++) {
        if (addList.get(i) == null || "".equals(addList.get(i))) continue;
        result[i] = new InternetAddress(addList.get(i));
    }
    return result;
}
}

```

MailSender 的 send 方法有一个 MailContext 和 Mail 参数, 只要能得到 MailContext 对象, 就可以得到发送邮件的 Session 对象, 得到 Mail 对象, 就可以得到邮件的所有信息, 包括主题、正文、抄送和附件等邮件相关信息, 最后将这些信息封装成一个 JavaMail 的 Message 对象, 使用 JavaMail 的 Transport 类进行发送即可。

## 12.8.2 界面封装Mail对象进行发送

实现了发送邮件的方法后, 就可以将界面的各个值封装成一个 Mail 对象, 再使用 MailSender 接口来进行发送即可, 以下是邮件编写界面 (MailFrame) 的发送邮件方法的实现。

代码清单: code\foxmail\src\org\crazyit\foxmail\ui\MailFrame.java

```

//发送方法
private void send() {
    if (!validateInput()) return;
    //得到 Mail 对象, 该对象表示原来的位置在已发送
    Mail mail = getMail(FileUtil.SENT);
    this.mailSender.send(mail, this.mainFrame.getMailContext());
}

```



```

        //添加了已发送的集合中
        this.mainFrame.addSentMail(mail);
        showMessage("你的邮件已成功发送");
        //清空界面组件的值
        clean();
        this.setVisible(false);
    }
    //将界面的组件封装成一个 Mail 对象
    private Mail getMail(String fromBox) {
        String xmlName = UUID.randomUUID().toString() + ".xml";
        Mail mail = new Mail(xmlName, getAddressList(this.receiver),
            this.mainFrame.getMailContext().getAccount(),
            this.subject.getText(), new Date(), "10",
            true, this.textArea.getText(), fromBox);
        mail.setCcs(getAddressList(this.cc));
        mail.setFiles(getFileListObjects());
        return mail;
    }
    //返回地址
    private List<String> getAddressList(JTextField field) {
        String all = field.getText();
        List<String> result = new ArrayList<String>();
        if (all.equals("")) return result;
        for (String re : all.split(",")) {
            result.add(re);
        }
        return result;
    }
    //从 JList 中得到附件的对象集合
    public List<FileObject> getFileListObjects() {
        ListModel model = this.fileList.getModel();
        List<FileObject> files = new ArrayList<FileObject>();
        for (int i = 0; i < model.getSize(); i++) {
            files.add((FileObject)model.getElementAt(i));
        }
        return files;
    }
}

```

以上代码中的 `send` 方法，从界面得到各个值后，封装成 `Mail` 对象，再使用 `MailSender` 接口进行发送。需要注意的是，在构造 `Mail` 对象的时候，`Mail` 对象的 `size` 属性值为 10，由于发送的时候并不需要关心该值的大小，当发送完成保存到已发送的 `Box` 时，就要设置这个 `size` 值。发送了邮件后，我们还需要将 `Mail` 对象保存到已发送的 `Box` 目录（`sent`），如果发送失败，那么就要保存到发件箱的 `Box` 目录中（`outbox`），以下代码实现这两个功能，为 `SystemHandler` 新增两个接口方法并予以实现。

代码清单：code\foxmail\src\org\crazyit\foxmail\system\impl\SystemHandlerImpl.java

```

    public void saveSent(Mail mail, MailContext ctx) {
        saveFiles(mail, ctx);
        //为 Mail 对象生成 xml 文件
        FileUtil.writeToXML(ctx, mail, FileUtil.SENT);
    }
    public void saveOutBox(Mail mail, MailContext ctx) {
        //保存 Mail 的附件
    }
}

```

```

        saveFiles(mail, ctx);
        FileUtil.writeToXML(ctx, mail, FileUtil.OUTBOX);
    }

```

那么就可以修改 `send` 方法，发送成功保存到已发送，失败保存到发件箱。

代码清单：code\foxmail\src\org\crazyit\foxmail\ui\MailFrame.java

```

//发送方法
private void send() {
    try {
        //省略其他代码
        this.systemHandler.saveSent(mail, this.mainFrame.getMailContext());
    } catch (Exception e) {
        //发送失败保存到发件箱
        this.systemHandler.saveOutBox(mail, this.mainFrame.getMailContext());
        showMessage(e.getMessage());
    }
}

```

### 12.8.3 处理邮件附件

处理邮件附件十分简单，提供一个文件选择器，让用户去选择文件后，再添加到附件的 `JList` 对象中即可，不过需要注意的是，当发送邮件时，从 `JList` 中取到的文件对象指向的是本地中的具体某份文件，当需要保存到发件箱、已发送的时候，就需要将这份具体的文件复制到 `file` 目录下，并且要将这份文件重命名，使用 `UUID` 生成新的文件名。上传完附件后，也可以进行删除附件的操作，只是从 `JList` 对象中删除该份文件对象即可。

在写邮件的过程中需要查看已上传的附件的话，与在主界面中查看附件有一点不同，并不需要另存为，由于这些附件是用户上传的，因此用户清楚该附件的位置，所以不需要另存为功能，那么我们可以继承 12.7.2 中的 `MainListMouseListener` 类，实现发送邮件的查看附件功能，新建 `SendListMouseListener` 类，以下为该类的具体实现。

代码清单：code\foxmail\src\org\crazyit\foxmail\ui\SendListMouseListener.java

```

public class SendListMouseListener extends MainListMouseListener {
    //重写父类的 handle 方法即可，只需要提供打开操作
    public void handle(FileObject file) {
        int result = JOptionPane.showOptionDialog(null, "请选择操作", "选择",
            0, JOptionPane.QUESTION_MESSAGE, null,
            new Object[]{"打开", "取消"}, null);
        if (result == 0) openFile(file);
    }
}

```

在编写邮件时，双击点击附件效果如图 12.9 所示。



图 12.9 编写邮件查看附件

### 12.8.4 测试发送邮件功能

邮件发送的功能已经全部实现，现在我们可以发送邮件进行测试，图 12.10 为发送邮件的效果图，运行程序，可以看到邮件已经发送。

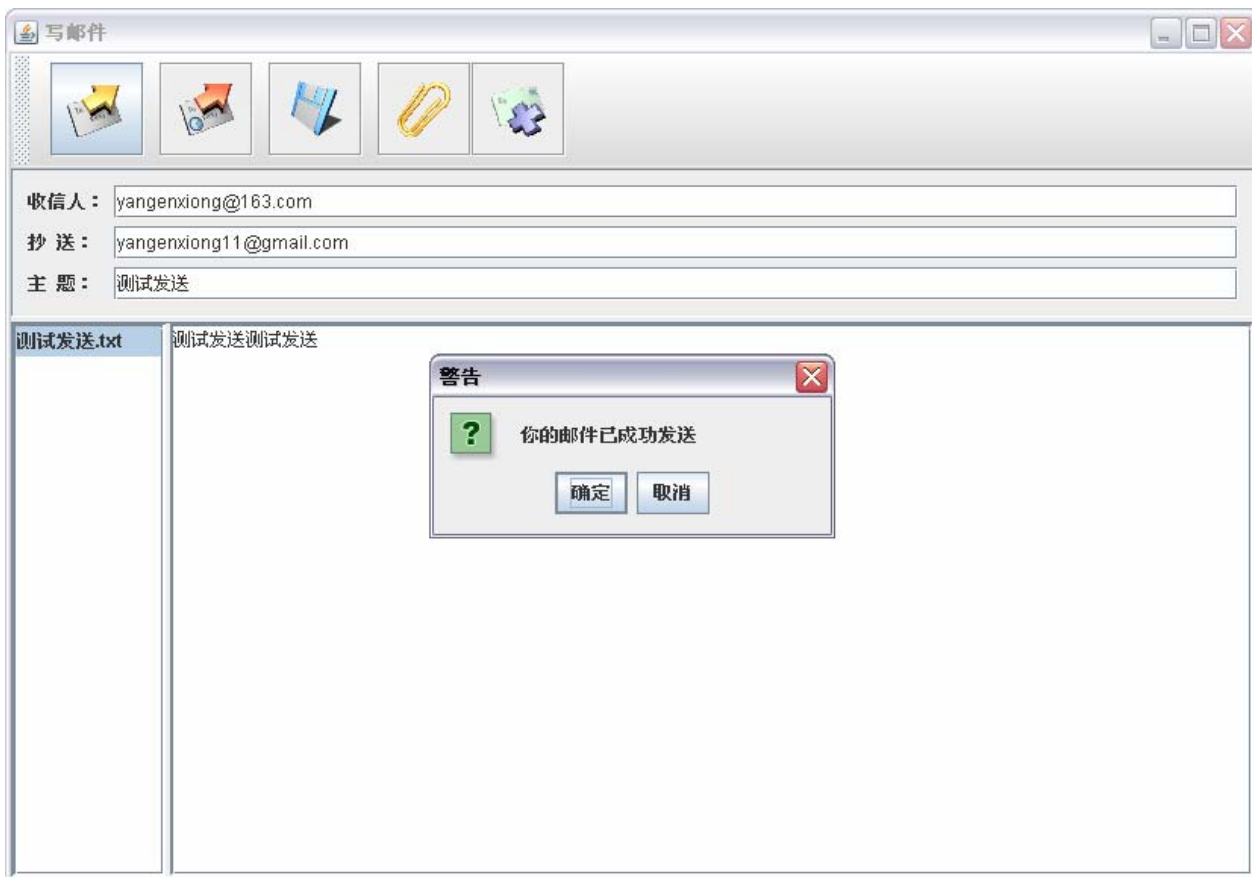


图 12.10 测试邮件发送

### 12.8.5 处理邮件的其他功能

邮件处理的其他功能包括回复邮件、转发邮件等，当用户选择了一封邮件进行回复操作时，可以将该邮件的相关内容设置到邮件的编写界面（MailFrame），回复的效果如图 12.11 所示。

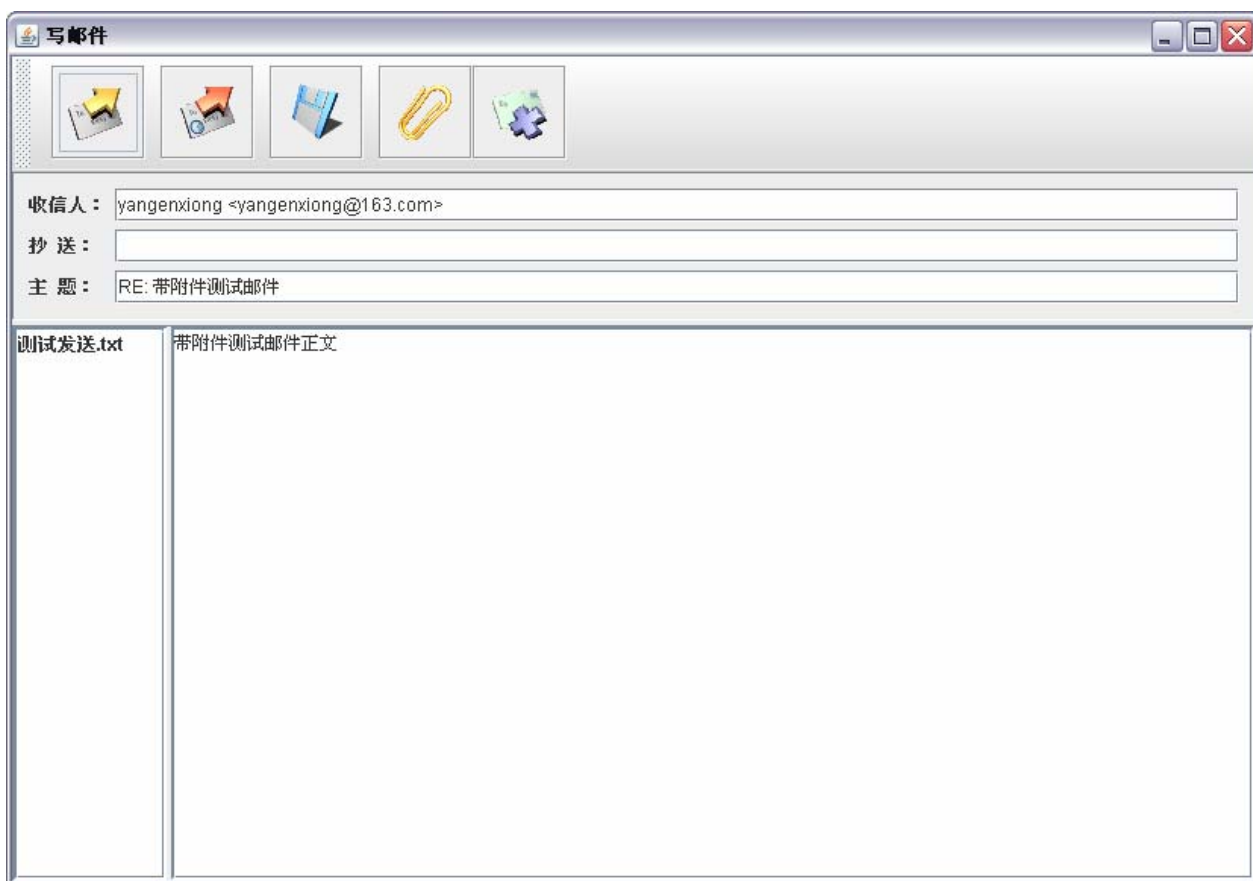


图 12.11 回复邮件

注意：回复邮件时，主题前需要加入“RE:”来表明这是回复的邮件。  
转发邮件原理也一样，只是收信人的信息并不能确定，具体转发给谁，由用户自己决定。

## 12.9 本章小节

本章中主要介绍了使用 **JavaMail** 进行收取和发送带附件的邮件，使用 **XStream** 进行对象和 **XML** 之间的转换，并将邮件对应的 **XML** 文件和邮件的附件保存到本地的相应目录中，在操作邮件的过程中，可以直接操作邮件对象，就实现了删除邮件、查看邮件和还原邮件等功能，本章案例尚有许多需要改进的地方，例如邮件的正文可以加入 **HTML** 转换，将一些 **HTML** 转换成可视的邮件正文，但由于这并不是本章的主要目的，因此在本例中没有提供实现。另外，在客户端收取邮件的时候，收取了多少邮件，也可以加入提醒的功能。如果需要使用本程序进行测试，建议使用 **Gmail** 的邮箱。希望本章的案例能对读者开发邮件应用有很好的启发。

## 第 13 章 MySQL 管理器

当前 IT 行业流行的关系型数据库有 Oracle、SQL Server、DB2 和 MySQL 等。其中 MySQL 是一个小型的关系型数据库，开发者是瑞典的 MySQL AB 公司，于 2008 年 6 月 1 日被 Sun 公司收购。MySQL 拥有体积小、速度快等优点，被广泛的应用在中小企业的 IT 系统中，更重要的一点是，它是开源的。由于 MySQL 应用广泛，因此涌现出许多 MySQL 的客户端，例如 MySQL Front、Navicat 与 MySQL 自带的 MySQL Administrator 等，这些都是我们平时在开发 MySQL 数据库应用时十分常用的 MySQL 图形管理工具。这些优秀的工具为我们提供了十分方便的功能去管理 MySQL 数据库，例如提供浏览数据的图形界面、操作数据的界面、操作各个数据库元素（表、视图、存储过程等）的界面，这些功能为我们带来了极大的方便，可以在一个完全图形化的界面进行数据库处理。使用这些工具，你可以不懂如何编写 SQL 语句，只需要通过操作图形界面，就可以达到操作数据库的目的。

在本章中，我们将自己开发一个简单的 MySQL 管理器。在开发的过程中，让大家了解前面所讲到的那些优秀工具的实现原理。在本章开头已经提到，这些管理工具，提供了各种的图形界面让我们去进行各种的操作，因此，开发一个 MySQL 管理器，除了需要编写一些操作数据库的 SQL 以外，还需要注意的是图形界面的处理。这些管理工具，实现的原理并无太大差别，但是哪个工具更能得到多数使用者的青睐，更多的就是取决于这些工具给用户带来的使用体验及方便性。

本章所开发的 MySQL 管理器是基于 MySQL5.0 开发的，因此如果要得到最佳的运行效果，请使用 MySQL5.0。由于 MySQL 各个版本间都存在差别，例如笔者在开发这个管理器的时候，就遇到 MySQL5.0 与 MySQL5.1 之间的微小差别，这些差别对我们开发所产生的影响，将在下面的章节中详细介绍。

### 13.1 MySQL 管理器原理

MySQL 管理器，主要功能是让用户可以轻松进行各种的 MySQL 操作，包括连接管理、数据库管理、表管理、视图管理、存储过程和函数管理，这些功能点我们都可以使用 JDBC 实现，例如表管理中包括创建表、修改表等功能，我们可以使用 JDBC 直接执行 SQL 语句中的 CREATE TABLE 和 ALTER TABLE 来达到目的。除了这些功能外，还需要对数据库中的数据进行导出和导入的操作，进行这些操作，我们可以编写程序来实现，但是，更好办法就是使用 MySQL 的命令（mysql 或者 mysqldump）来解决，这样可以轻松解决数据的导出与导入，但是，前提就是使用的客户端必须安装 MySQL 数据库，并且要告诉我们这个管理器，MySQL 的具体目录，我们可以使用程序去调用这些 MySQL 的命令。下面，我们就开始实现这些所定义的功能。

### 13.2 建立界面

在编写程序前，我们需要准备各个界面，包括连接管理界面、表管理界面、视图管理界面、存储过程（函数）管理界面与查看数据界面等。表管理、视图管理、存储过程和函数管理我们可以建立一个主界面，根据不同的情况显示不同的菜单，而连接管理我们可以使用一棵树来进行管理，可以同时存在多个连接，这些连接下面的子节点就是该连接下面的数据库。

### 13.2.1 MySQL 安装目录选择界面

当进入管理器时，我们就需要让用户去选择 MySQL 的安装目录，原因就是因为我们需要 MySQL 的内置命令，因此需要指定 MySQL 的安装目录。图 13.1 是安装目录选择界面。



图 13.1 MySQL 安装目录选择界面

让用户选择 MySQL 安装目录十分简单，只提供一个目录选择安装以及显示目录路径的 `JTextFeild`，并且加入一个确定与取消按钮。当用户选择了 MySQL 的安装目录，点击了确定时，就显示我们的主界面，这里需要注意的是，我们在实现的时候，需要判断用户所选择的 MySQL 安装目录是否正确，由于 `mysql` 与 `mysqldump` 等命令是存在于 MySQL 安装目录下的 `bin` 目录的，因此判断用户所选择的目录是否正确，可以判断在 `bin` 目录下是否存在相应的命令，这些将在下面的章节中描述。MySQL 安装目录在本章代码中对应的是 `ConfigFrame` 类。

### 13.2.2 主界面

主界面提供各种功能的入口，可以让用户在该界面中使用或者进入各个功能，除了需要提供这些入口外，还需要提供一棵树，更直观的展示当前所使用的连接，以及该连接下面所有的数据库。主界面如图 13.2 所示。

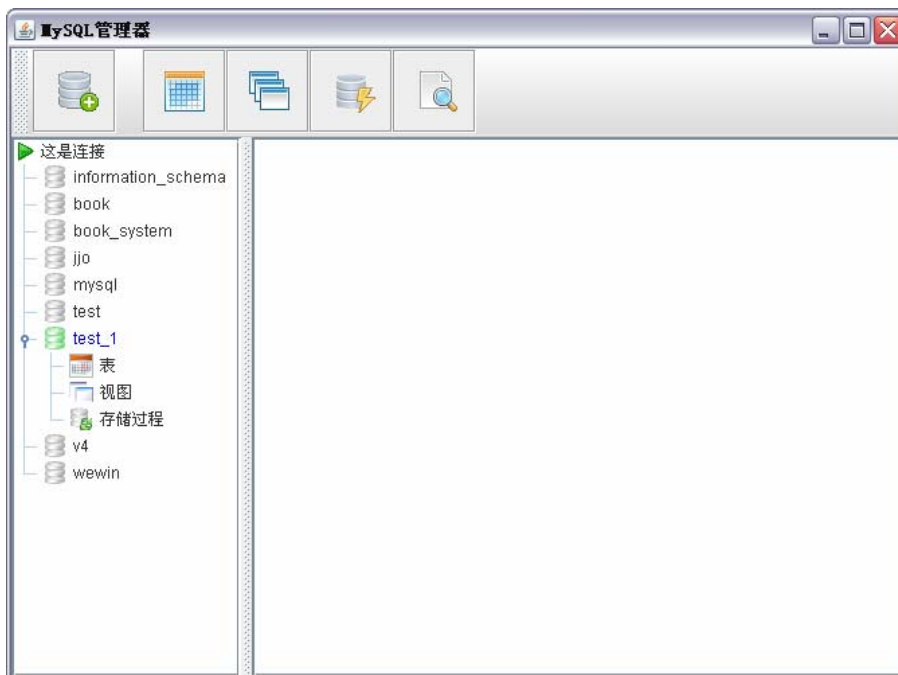


图 13.2 主界面

主界面由一个工具栏，一棵树以及一个 **JList** 组成，其中工具栏中包括的操作如下：

- ☐ 添加连接：可以让用户添加一个连接。
- ☐ 查看表：查看当前数据库中所有的表。
- ☐ 查看视图：查看当前数据库中所有的视图。
- ☐ 查看存储过程（函数）：查看数据库中所有的存储过程与函数。
- ☐ 打开执行 **SQL** 的界面：打开一个执行 **SQL** 语句的界面。

在主界面的左边部分，提供一棵树让用户十分直观的看到连接的相关信息，这棵树可以隐藏根节点，第一层节点就是连接，第二层节点就是该连接下所对应的所有的数据库，每一个数据库节点下面可以有三个子节点：表、视图和存储过程，当然，我们在平时使用其他管理工具的时候，还可以有触发器等内容，我们在本章的项目中不提供这些功能。

这里需要注意的是，我们更换了树的各个节点图片，因此需要为 **JTree** 添加一个 **DefaultTreeCellRenderer** 来设置各个节点的图片以及文字，当然，还需要将各个节点抽象成不同的对象，新建各个视图对象的接口 **ViewObject**，该接口将是所有视图对象的父类，这些视图对象包括树的各个节点，主界面右边列表所显示的各个元素等。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\ViewObject.java

```
public interface ViewObject {
    //返回显示的图片
    Icon getIcon();
}
```

该接口只有一个 **getIcon** 方法，返回一个 **Icon** 对象，表示这些界面所对应的图片，另外，树上的各个节点对象，可以有两种形式，第一种就是需要带连接的节点，例如数据库连接节点和数据库节点，第二种就是不需要带有连接的节点，因此我们可以将带有连接的节点抽象成一个父类，让连接节点和数据库节点去继承。另外，还需要提供一个 **connect** 的抽象方法，需要让子类去实现。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\tree\ConnectionNode.java

```
public abstract class ConnectionNode implements ViewObject {
    //JDBC 的 Connection 对象
    protected Connection connection;
    //连接方法，由子类去实现
    public abstract Connection connect();
    //省略 setter 和 getter 方法
}
```

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\tree\ServerConnection.java

```
public class ServerConnection extends ConnectionNode {
    private final static String DRIVER = "com.mysql.jdbc.Driver";//MySQL 驱动
    private String connectionName; //MySQL 驱动
    private String username; //用户名
    private String password; //密码
    private String host; //连接 ip
    private String port; //连接端口
    //省略 setter 和 getter 方法
    //实现接口 ViewObject 的方法，根据不同的连接状态显示不同的图片
    public Icon getIcon() {
        if (super.connection == null) return ImageUtil.CONNECTION_CLOSE;
        else return ImageUtil.CONNECTION_OPEN;
    }
    //重写 toString 方法，返回连接名称
    public String toString() {
        return this.connectionName;
    }
}
```

```

    }
    //实现父类的 connect 方法
    public Connection connect() {
    }
}

```

一个 **ServerConnection** 对象表示一个连接节点，一个连接节点当然需要包括一些连接的相关信息，包括连接名称、MySQL 用户名、密码、连接的 IP 与端口等信息。该对象实现了 **ViewObject** 的 **getIcon** 方法，判断父类 **ConnectionNode** 的 **connection** 属性是否为空来显示不同的图片，还需要重写 **toString** 方法，返回连接的名称。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\tree\Database.java

```

public class Database extends ConnectionNode {
    private String databaseName; //数据库名字
    private ServerConnection serverConnection; //数据库所属的服务器连接
    //需要使用数据库名称与服务器连接对象构造
    public Database(String databaseName, ServerConnection serverConnection) {
        this.databaseName = databaseName;
        this.serverConnection = serverConnection;
    }
    //实现接口的方法，判断该数据库是否连接，再返回不同的图片
    public Icon getIcon() {
        if (this.connection == null) return ImageUtil.DATABASE_CLOSE;
        return ImageUtil.DATABASE_OPEN;
    }
    //重写 toString 方法
    public String toString() {
        return this.databaseName;
    }
    //实现父类的 connect 方法
    public Connection connect() {
    }
}

```

**Database** 节点对象包括数据库的名字，另外还需要一个 **ServerConnection** 对象，由于每个数据库都是某个连接节点下面的子节点，因此需要记录它的父节点，当然，并不是简单的进行记录，还可以让它们共享一些不会经常创建的实例，例如 **Connection**。另外，需要注意的是，无论 **ServerConnection** 或者 **Database** 对象，都需要实现 **ViewObject** 的 **getIcon** 方法，当连接节点或者数据库节点被打开时，都需要改变它们的图片，而显示何种图片，由 **getIcon** 方法决定。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\tree\TableNode.java

```

public class TableNode implements ViewObject {
    private Database database; //所属的数据库节点
    //返回表的树节点图片
    public Icon getIcon() {
        return ImageUtil.TABLE_TREE_ICON;
    }
    //重写 toString 方法
    public String toString() {
        return "表";
    }
}

```

一个 **TableNode** 对象代表一个表的节点，需要提供一个 **Database** 属性来表示这个对象是属于哪个



数据库下面的子节点。如图 13.2 所示，我们可以看树中每个数据库节点的表节点都是一致的（每个数据库里面都有表），可以将这个表节点理解成是导航栏的某一组成部分，当用户点击了这个节点后，就可以在右边的列表中显示对应数据库的表。

与 `TreeNode` 一样，另外再次创建两个对象：`ViewNode` 和 `ProcedureNode`，分别代表数据库节点下面的视图节点和存储过程节点，实现方法与 `TreeNode` 类似。下面为树节点添加一个 `DefaultTreeCellRenderer` 类，让其得到这些节点对象，并设置相应的文字和图片。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\tree\TreeCellRenderer.java

```
public class TreeCellRenderer extends DefaultTreeCellRenderer {
    public Component getTreeCellRendererComponent(JTree tree, Object value,
        boolean sel, boolean expanded, boolean leaf, int row,
        boolean hasFocus) {
        DefaultMutableTreeNode node = (DefaultMutableTreeNode)value;
        //获得每个节点的 ViewObject
        ViewObject obj = (ViewObject)node.getUserObject();
        if (obj == null) return this;
        this.setText(obj.toString()); //设置文字
        this.setIcon(obj.getIcon()); //设置图片
        if (sel) this.setForeground(Color.blue); //判断是否选来设置字体颜色
        else this.setForeground(getTextNonSelectionColor());
        return this;
    }
}
```

在节点处理类 `TreeCellRenderer` 类，得到每个节点的 `ViewObject` 后，就可以为节点设置文字和图片，我们的 `ViewObject` 接口提供了 `getIcon` 方法，所以我们就可以在节点处理类中得到每个节点所对应的图片与文字（从 `toString` 方法获得）。

树的相关处理就完成了，主界面的右边是一个列表，对应的是一个 `JList` 对象，`JList` 里面的每一个元素，都是 `ViewObject` 的实现类，只需要实现 `getIcon` 方法与重写 `toString` 方法即可。每个列表的元素对象都可以将它们的 `name` 属性抽象到一个父类中，各个对象去继承它即可，在本例中，我们所涉及有三种数据类型：表、视图和存储过程（函数），我们需要建立三个对象，分别代表这三种数据类型。

在本章的代码中，我们创建了 `TableData`、`ViewData` 和 `ProcedureData` 三个类分别代表表数据、视图数据和存储过程数据，这三个对象都需要实现 `ViewObject` 接口，具体的实现与三个节点的实现类似，都需要实现 `getIcon` 方法并重写 `toString`。表、视图和存储过程都是某一个数据库下面的元素，因此这三个数据对象都需要保存一个 `Database` 属性，表示该数据所属于的数据库。与树一样，还需要提供一个元素处理类，来指定显示的数据图片。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\list>ListCellRenderer.java

```
public class ListCellRenderer extends DefaultListCellRenderer {
    public Component getListCellRendererComponent(JList list, Object value,
        int index, boolean isSelected, boolean cellHasFocus) {
        JLabel label = (JLabel)super.getListCellRendererComponent(list,
            value, index, isSelected, cellHasFocus);
        ViewObject vd = (ViewObject)value; //得到 ViewObject 对象
        label.setIcon(vd.getIcon()); //设置图片
        label.setToolTipText(vd.toString());
        //设置选中时的字体颜色
        if (isSelected) {
            setBackground(Color.blue);
            setForeground(Color.white);
        }
        return this;
    }
}
```

```
}  
}
```

到这里，主界面的各个对象都创建好了，本章中对应的主界面对象是 **MainFrame** 类，可以在该类中创建对应的树与列表。这里需要注意的是，当创建列表（**JList**）的时候，可以将 **JList** 设置为横向滚动，调用以下代码即可实现：

```
dataList.setLayoutOrientation(JList.VERTICAL_WRAP); //dataList 是界面中的 JList 对象
```

创建主界面后，我们可以在创建树与创建列表的时候加入一些模拟数据来查看效果，具体的效果如图 13.3 所示：

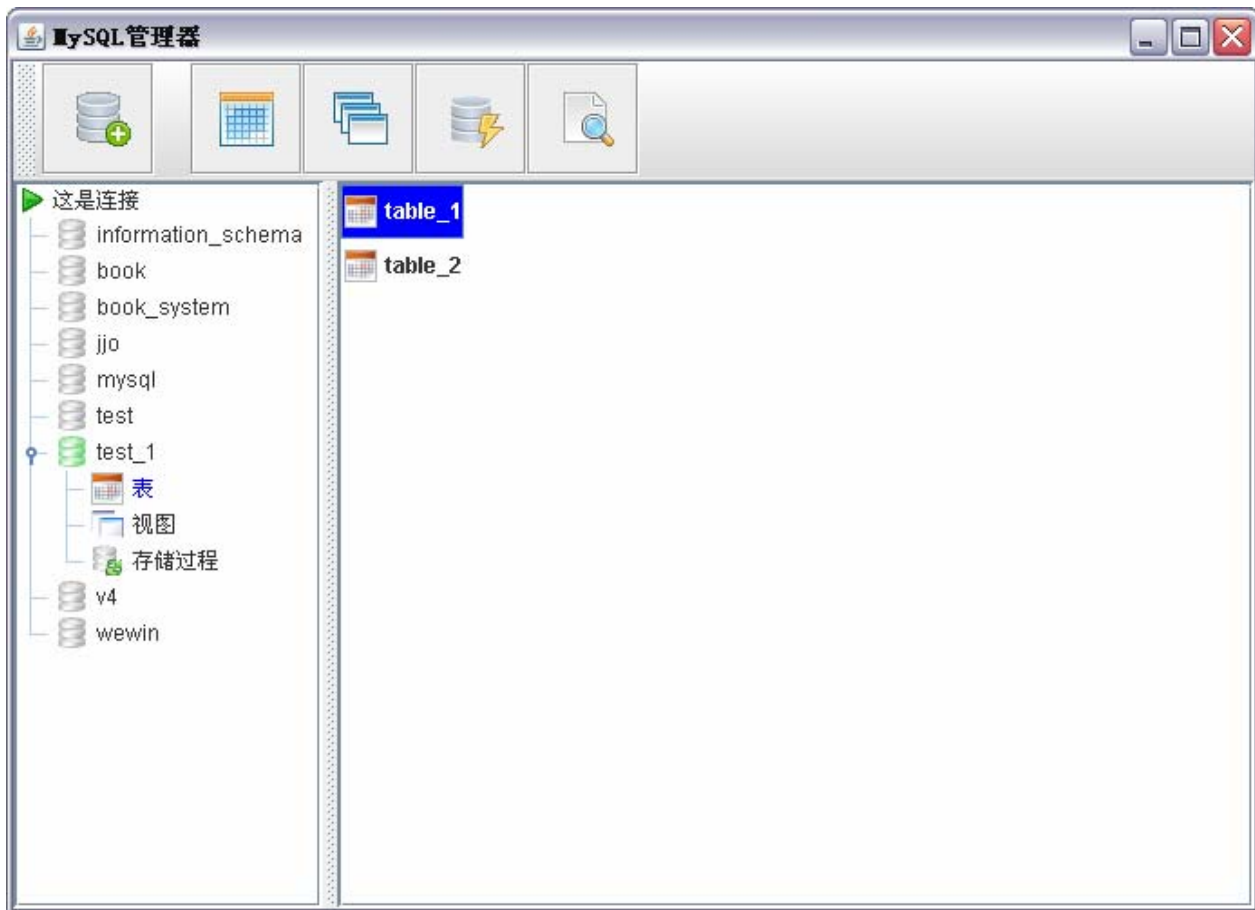
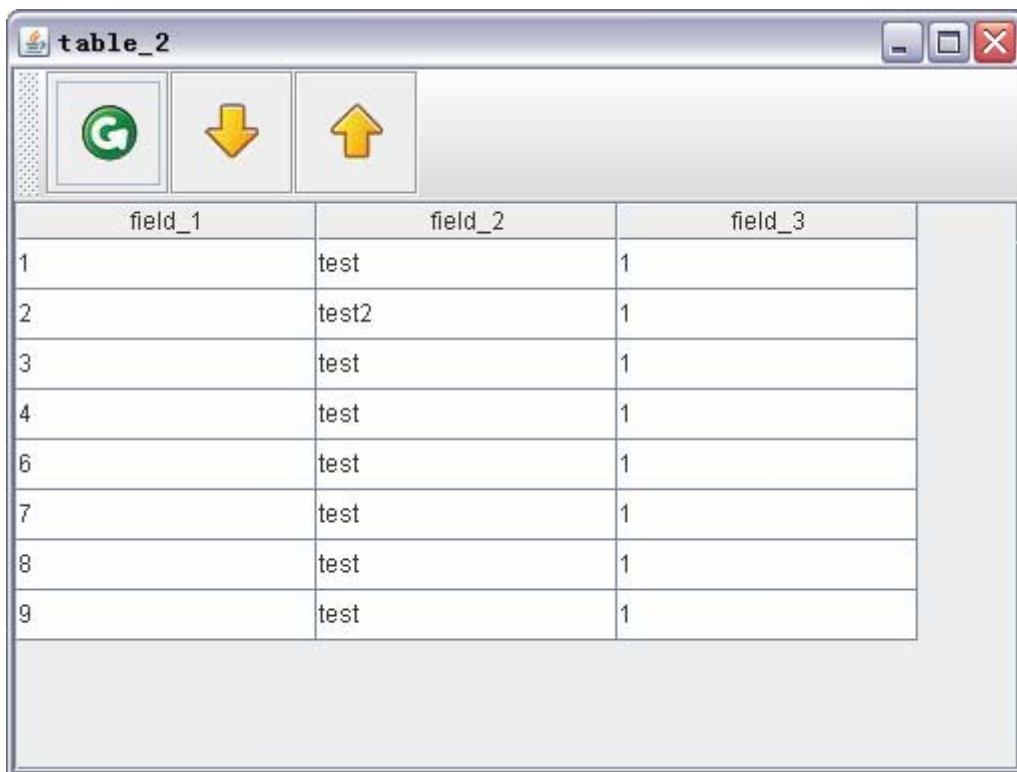


图 13.3 主界面效果

### 13.2.3 数据显示界面

在整个管理器中，我们需要一个数据显示的界面，而且只有一个。打开数据显示界面的途径有两种，一种是双击一个表查看数据的时候，另外一种就是执行 **SQL** 的时候（执行查询的 **SQL**），就会打开数据显示界面，将用户感兴趣的数据显示出来。由于一般会存在打开多个表或者多次执行 **SQL** 的情况，因此我们在编写打开数据显示界面的代码的时候，每次都需要去创建这个界面对象的实例。在本章中，界面显示对象对应的类是 **DataFrame**，数据显示界面如图 13.4 所示。



field_1	field_2	field_3
1	test	1
2	test2	1
3	test	1
4	test	1
6	test	1
7	test	1
8	test	1
9	test	1

图 13.4 数据显示界面

界面比较简单，一个工具条加一个表格即可，工具条中包括的功能有：

- ❑ 刷新：刷新当前界面的数据。
- ❑ 降序：当用户选择了某一列并点击该图标的时候，就对该列所对应的字段进行降序排序。
- ❑ 升序：操作与降序一样，但是对所选字段进行升序排序。

在这个界面中，需要注意的是，这个列表对应的 `JTable` 对象并不像其他 `JTable` 一样，拥有固定的列，由于我们不可能知道用户将要打开的表有多少列，因此只能在用户打开表的时候，得到该表的信息再动态的生成列与数据。除了这里之外，我们还需要为这个 `JTable` 对象进行一些额外的处理，例如我们需要让这个 `JTable` 对象可以整列选择，就需要自己编写一个类去继承 `JTable`。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\table\DataTable.java

```
//当点击表头时，表示当前所选择的列
private int selectColumn = -1;
public DataTable(DefaultTableModel model) {
    //为表头添加鼠标事件监听器
    header.addMouseListener(new MouseAdapter() {
        public void mouseClicked(MouseEvent e) {
            header.getTable().clearSelection();
            int tableColumn = header.columnAtPoint(e.getPoint());
            selectColumn = tableColumn;
        }
    });
    //为 JTable 添加鼠标监听器
    this.addMouseListener(new MouseAdapter() {
        public void mouseClicked(MouseEvent e) {
            selectColumn = -1;
            updateUI();
        }
    });
}
```

```
    }  
    });  
}
```

注意以上代码中的类属性 `selectColumn`，当我们用鼠标点击了表头的时候，就将该值设为当前选择的列的索引，当在 `JTable` 的其他地方点击了鼠标时，就设置该值为-1，表示没有选择表头。那么我们就需要重写 `JTable` 的 `isCellSelected` 方法，如果 `selectColumn` 不是-1，那么就需要将用户所选择的列整列设为选中状态，以下是 `isCellSelected` 方法的实现：

```
//判断一个单元格是否被选中，重写 JTable 的方法  
public boolean isCellSelected(int row, int column) {  
    if (this.selectColumn == column) return true; //如果列数与当前选择的列相同,返回 true  
    return super.isCellSelected(row, column);  
}
```

另外，我们还需要提供一个返回 `selectColumn` 值的 `public` 的方法。做完这些后，可以点击一列，看到效果如图 13.5 所示。

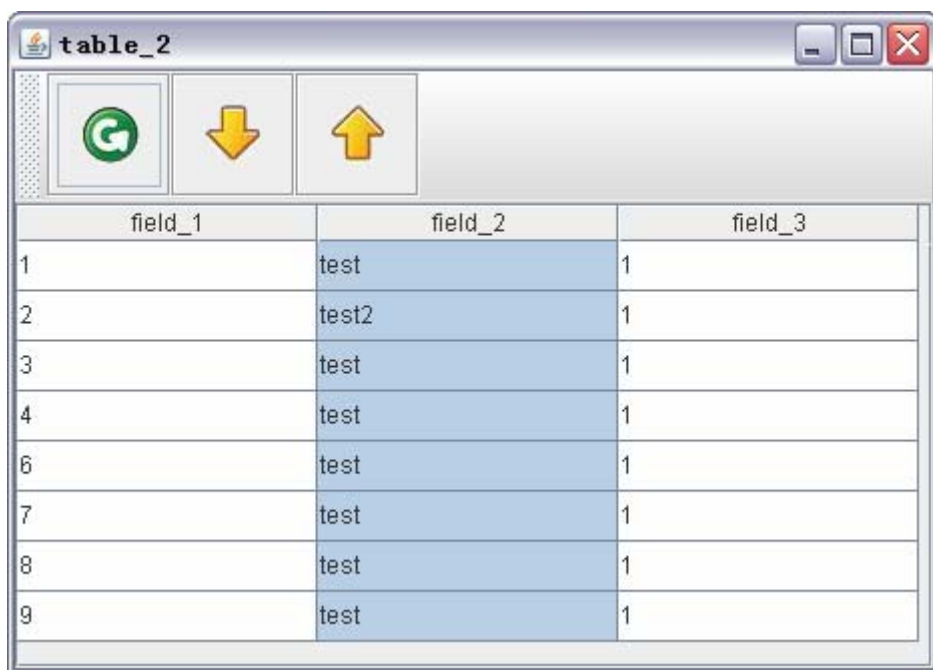


图 13.5 数据显示界面选择整列

### 13.2.4 创建连接界面

连接是整个工具的最基础部分，没有连接，其他任何操作都不能进行，因此使用这个 MySQL 管理工具，就需要提供一个新增连接的界面，让用户去创建各个连接，界面如图 13.6 所示。



图 13.6 新建连接界面

图 13.6 中新建连接的界面比较简单，普通的一个表单，界面中包括的元素如下：

- ☐ 连接名称：该名称在管理器的树中显示，并且该名称不可以重复。
- ☐ 连接 IP：需要连接到的 MySQL 服务器 IP。
- ☐ 端口：MySQL 的端口，默认为 3306。
- ☐ 用户名：连接 MySQL 的用户名，例如 root。
- ☐ 密码：连接 MySQL 的密码。
- ☐ 测试连接：测试输入的信息是否可以连接到 MySQL 服务器中，当然，如果测试不能连接，也可以添加这个连接。
- ☐ 确定和取消：点击确定添加连接并关闭该窗口，点击取消不保存连接并关闭窗口。

### 13.2.5 创建表界面

当用户需要创建一个表的时候，就需要提供一个界面让用户去输入表的各种数据，包括字段名称、类型、是否允许空和主键等信息。创建表界面是本章中最为复杂的界面，用户可以随意的在表中进行操作，最后执行保存，表界面如图 13.7 所示。

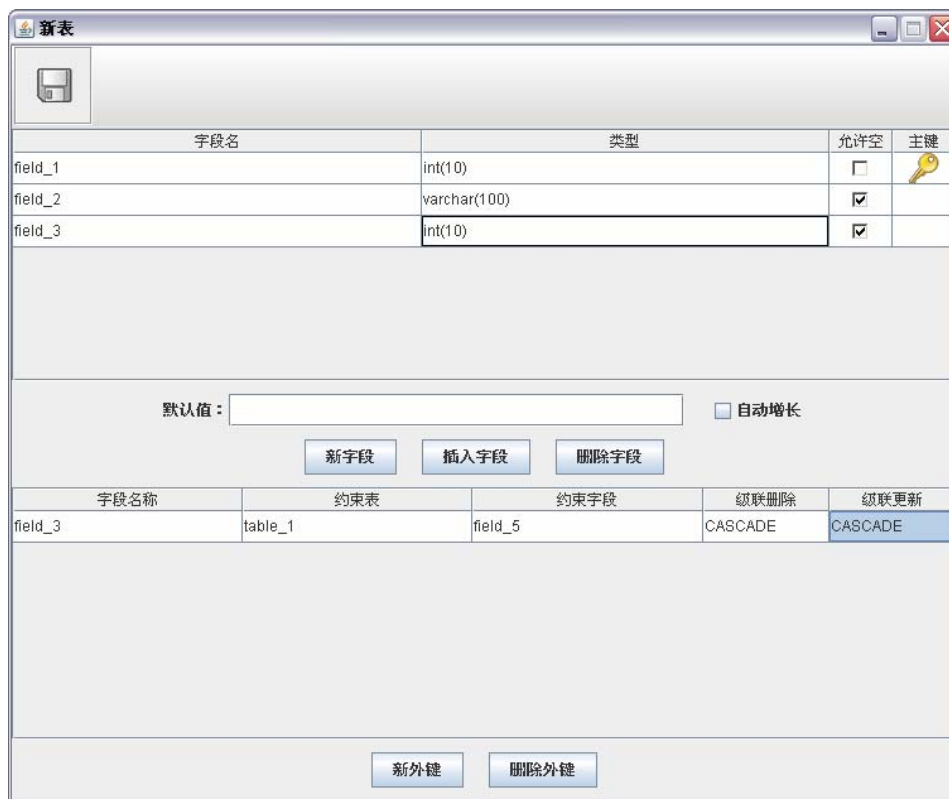


图 13.7 创建表界面

界面如图 13.7 所示，该界面较为复杂，分成上下两个表格，上面的表格主要处理表的字段信息，包括字段名、类型、是否允许空和主键，在该表格下面，有一个输入默认值的文本框，并提供一个表示字段是否自动增长的多选框。当我们在表格中选某行数据（字段）的时候，默认值就需要发生相应的改变，自动增长的多选框也要随着改变。在本章中表界面对应的是 **TableFrame** 类。

字段表格需要进行特别处理的是允许空和主键的单元格，这两个单元格都需要使用图片来显示。我们编写一个 **FieldTable** 类来表示字段表格，并为这个 **FieldTable** 提供一个 **DefaultTableCellRenderer** 的子类来对单元格进行处理。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\table\FieldTableIconCellRenderer.java

```
public class FieldTableIconCellRenderer extends DefaultTableCellRenderer {
    public Component getTableCellRendererComponent(JTable table, Object value,
        boolean isSelected, boolean hasFocus, int row, int column) {
        //判断单元格的值类型，分别调用 setIcon 与 setText 方法
        if (value instanceof Icon) this.setIcon((Icon)value);
        else this.setText((String)value);
        this.setHorizontalAlignment(CENTER);
        return this;
    }
}
```

**FieldTableIconCellRenderer** 的实现十分简单，只是判断单格的值再进行处理。在 **FieldTable** 使用以下代码即可实现显示图片。

```
this.getColumn(ALLOW_NULL).setCellRenderer(this.cellRenderer);
this.getColumn(PRIMARY_KEY).setCellRenderer(this.cellRenderer);
```

以上代码先得到允许空和主键的列后再设置单元格处理类。重新运行程序时，就可以看到效果如图 13.7 所示，但是否需要对 **FieldTable** 加入鼠标事件处理，当点击了允许空和主键的列单元格时，就需要改变它们图片。为 **FieldTable** 加入鼠标监听器。

代码清单: code\mysql-manager\src\org\crazyit\mysql\ui\table\FieldTable.java

```
//鼠标在 JTable 中点击的时候触发该方法
private void selectCell() {
    int column = this.getSelectedColumn();
    int row = this.getSelectedRow();
    if (column == -1 || row == -1) return;
    //修改图片列
    selectAllowNullColumn(row, column);
    selectPrimaryKeyColumn(row, column);
}

//点击的单元格位于允许空列
private void selectAllowNullColumn(int row, int column) {
    //得到需要更改图片的列（允许空列）
    TableColumn tc = this.getColumn(ALLOW_NULL);
    if (tc.getModelIndex() == column) {
        Icon currentIcon = (Icon)this.getValueAt(row, column);
        //根据当前选中的图片来更改允许空的图片
        if (ImageUtil.CHECKED_ICON.equals(currentIcon)) {
            this.setValueAt(ImageUtil.UN_CHECKED_ICON, row, column);
        } else {
            this.setValueAt(ImageUtil.CHECKED_ICON, row, column);
        }
    }
}

//如果鼠标点击的列是"主键列", 去掉或者加上图标
private void selectPrimaryKeyColumn(int row, int column) {
    //得到需要更改图片的列（主键列）
    TableColumn tc = this.getColumn(PRIMARY_KEY);
    if (tc.getModelIndex() == column) {
        Object obj = this.getValueAt(row, column);
        if (ImageUtil.PRIMARY_KEY_BLANK.equals(obj)) {
            this.setValueAt(ImageUtil.PRIMARY_KEY, row, column);
        } else {
            this.setValueAt(ImageUtil.PRIMARY_KEY_BLANK, row, column);
        }
    }
}
}
```

只需要在创建 **FieldTableIconCellRenderer** 的时候为表格加入鼠标监听器, 该监听器调用以上代码的 **selectCell** 方法即可, **selectCell** 方法再去调用点击允许空和主键单元格的方法, 即以上的 **selectAllowNullColumn** 和 **selectPrimaryKeyColumn** 方法, 这两个方法中判断用户所选择的列, 是否需要进行图片处理的列（允许空和主键）, 再对单元格的值（图片）进行修改, 就可以达到点击单元格就显示不同图片的效果。另外, 当我们点击了某行数据（字段）的时候, 还需要处理默认值与自动增长, 我们在下面章节将会实现。

实现了字段列表后, 还需要注意的是该列表下面的三个按钮, 分别是新字段、插入字段和删除字段, 新字段与插入字段的区别是, 新字段在列表的最后加入一行数据, 插入字段在用户所选择的行的前面插入一行数据。

**TableFrame** 下面的外键列表与字段列表不同的是, 外键列表不需要进行图片处理, 但是每个单元格都需要使用下拉框来代替普通的文字。与字段列表一样, 新建一个 **ForeignTable** 的类来表示一个外键列表, 外键列表有 5 列, 而且每一列中的每个单元格都是下拉框, 因此我们需要在 **ForeignTable** 中创

建 5 个下拉框 (JComboBox) 以及 5 个单元格编辑器对象。

5 个单元格编辑器对象, 以下是 ForeignTable 的实现。

代码清单: code\mysql-manager\src\org\crazyit\mysql\ui\table\ForeignTable.java

```
private DefaultCellEditor fieldNameEditor; //字段名称编辑器对象
private DefaultCellEditor referenceTableEditor; //约束表
private DefaultCellEditor referenceFieldEditor; //约束字段
private DefaultCellEditor onDeleteEditor; //级联删除
private DefaultCellEditor onUpdateEditor; //级联更新
```

那么在创建这些单元格编辑器对象的时候, 就分别以各个下拉框的对象作为构造参数:

```
this.fieldNameEditor = new DefaultCellEditor(this.fieldNameComboBox);
```

接下来, 得到相应的列, 再设置编辑器对象即可:

```
this.getColumnModel().getColumn(FIELD_NAME).setCellEditor(this.fieldNameEditor);
```

做完这些工作后, 外键列表中所有的单元格都变成可以使下拉来设定值, 我们在开发界面的时候, 由于缺乏真实的数据, 因此我们可以提供一些模拟的数据来实现效果, 到需要实现的时候, 就可以替换上真实的数据。新增表与修改表的界面可以共用一个界面, 但是同时需要做新增与修改操作的时候, 就需要做多一些额外的判断, 本章中新增表与修改表为同一个界面 (TableFrame)。

### 13.2.6 视图界面

当用户需要编写一个视图的时候, 我们可以提供一个视图界面。视图界面实现十分简单, 只有一个 JTextArea 即可, 并附带有保存操作。这里需要注意的是, 用户点击保存的时候, 需要将视图通过 SQL 的 CREATE VIEW 来创建, 那么用户查看视图的时候, 与查看表一样, 都是需要打开数据浏览界面。图 13.8 是视图界面。

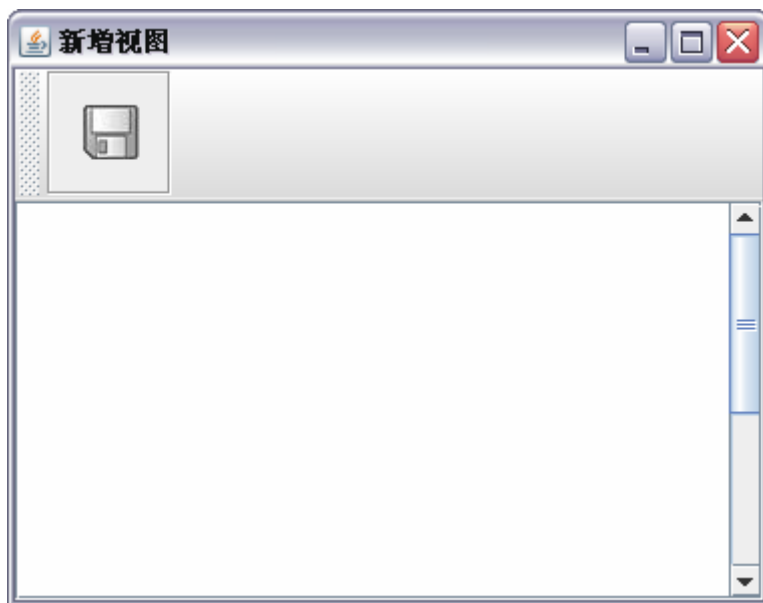


图 13.8 视图界面

在本章中, 创建表的界面一样, 无论新增视图或者修改视图, 都使用相同的一个界面, 对应的是 ViewFrame。



### 13.2.7 存储过程界面

用户需要新建一个存储过程或者函数的时候，可以提供一个新建存储过程界面让用户去操作。存储界面在本章中对应的类是 `ProcedureFrame`。存储过程界面如图 13.9 所示。

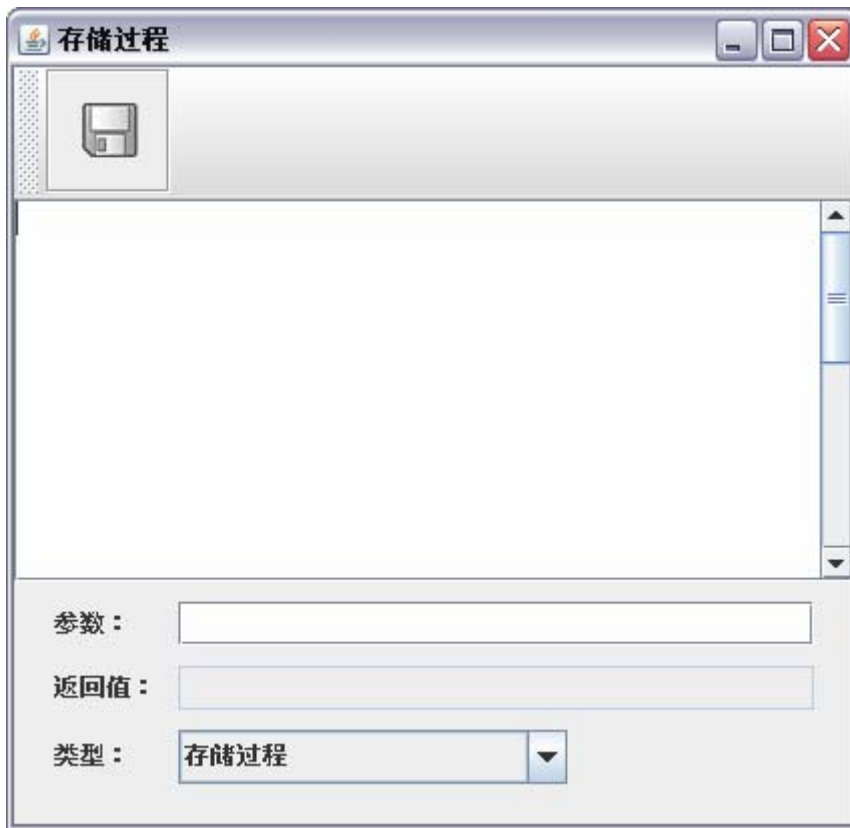


图 13.9 存储过程界面

界面元素说明：

- ❑ 输入方法体的 `JTextArea`：用户可以在此输入存储过程或者函数的方法体。
- ❑ 参数 `JTextField`：输入存储过程或者函数的参数。
- ❑ 返回值 `JTextField`：可以输入函数的返回值，因为函数才有返回值。如果选择的类型为存储过程，则该 `JTextField` 不可用。
- ❑ 类型下拉框：可以选择编写的类型，是存储过程还是函数。

### 13.2.8 查询界面

当用户需要执行一些 SQL 的时候，可以提供一个查询界面让用户去输入，该界面提供执行 SQL 与保存 SQL 的功能，执行 SQL 的时候，如果是普通的 `INSERT`、`UPDATE` 或者其他无需浏览数据的 SQL 语句，则可以直接操作。如果执行的是查询、调用存储过程或者函数的语句，那么就需要将结果显示到数据界面，即 13.2.3 的界面。本章对应的查询界面类是 `QueryFrame`，查询界面如图 13.10 所示。

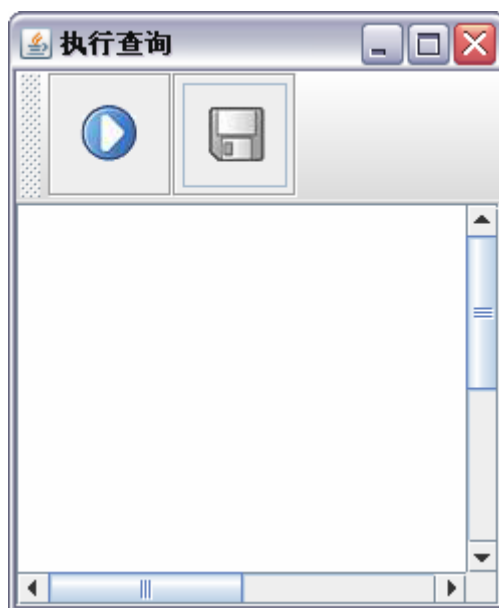


图 13.10 查询界面

### 13.2.9 树节点右键菜单

在主界面的连接树中，当我们点击了树的某个节点的时候，可以提供一些右键菜单来执行一些相关的操作，例如点击了连接节点，就可以提供关闭连接、删除连接等右键菜单，如果点击了数据库节点，就可以提供关闭数据库或者删除数据库等右键菜单。

点击连接节点的右键菜单如图 13.11 所示。

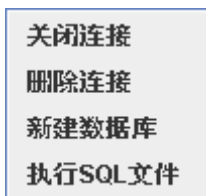


图 13.11 连接节点菜单

点击数据库节点的右键菜单如图 13.12 所示。

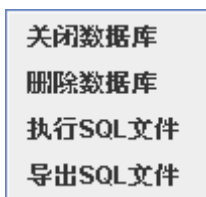


图 13.12 数据库节点右键菜单

由于我们对连接节点或者数据库节点进行选择的时候，就可以打开连接或者数据库，因此并不需要提供打开的菜单，本章中使用 `JPopupMenu` 来实现鼠标右键菜单，`MainFrame` 中提供一个 `JPopupMenu` 对象来存放各个菜单当点击了连接节点的时候 `JPopupMenu` 删除所有的子菜单，再加入连接节点的菜单（`JMenuItem`），数据库节点的实现方式与之相同。

### 13.2.10 数据列表右键菜单

主界面中除了连接树外，还有一个数据列表，当用户在树中点击了表节点、视图节点或者存储过程节点的时候，数据列表中就显示不同的数据，我们可以根据当前所显示的数据来创建不同的鼠标右键菜单。图 13.13 是数据列表显示表数据的时候的右键菜单。



图 13.13 表数据菜单

表数据鼠标右键菜单说明：

- ☐ 新建表：打开创建表的界面，即 13.2.5 中的界面。
- ☐ 编辑表：修改一个表，与新建表使用同一个界面。
- ☐ 删除表：删除列表中选择数据。
- ☐ 导出表：将一个表的数据导出。

视图数据的鼠标右键菜单如图 13.14 所示。



图 13.14 视图数据菜单

视图数据鼠标右键菜单说明：

- ☐ 新建视图：打开 13.2.6 中的视图界面，用于创建视图。
- ☐ 编辑视图：修改所选择的视图，与新建视图使用同一个界面。
- ☐ 删除视图：删除所选择的视图。

存储过程鼠标右键菜单如图 13.15 所示。



图 13.15 存储过程数据菜单

存储过程鼠标右键菜单说明：

- ❑ 新建存储过程：打开 13.2.7 中的存储过程界面，创建存储过程。
- ❑ 编辑存储过程：修改选择的存储过程，与新建存储过程使用相同的界面。
- ❑ 删除存储过程：删除所选择的存储过程或者函数。

以上为三种数据的右键菜单，实现方式与树节点的右键菜单一样，当界面的数据发生改变时，就相应的去删除 JPopupMenu 所有的子菜单，再添加相应的菜单（JMenuItem）即可。

以上的菜单均在主界面（MainFrame）中创建，程序并不知道当前显示的是哪种数据，因此我们需要在 MainFrame 中提供一个 ViewObject 的类来标识当前显示的类型，ViewObject 是所有界面元素都需要实现的接口，表数据是 TableData 类，视图数据是 ViewData 类，存储过程数据是 ProcedureData 类，详细请看 13.2.2 中的各个界面对象。当用户点击了工具栏或者树上的某个节点时，就相应的改变 MainFrame 中的 ViewObject 即可。

到此，管理器的所有界面都创建完毕，接下来就可以实现相关的功能。

## 13.3 实现MySQL安装目录选择功能

实现 MySQL 安装目录选择功能，我们使用 13.2.1 的界面。当用户进入管理器的时候，就让用户选择本地的 MySQL 安装目录，由于我们需要使用 MySQL 的一些内置命令，因此选择 MySQL 的安装目录是一个必要的操作，得到 MySQL 安装目录后，我们就可以找到 bin 目录下面的命令。因此用户选择了安装目录后，我们的程序就需要对所选择目录进行验证，判断能否找到 bin 目录。

### 13.3.1 实现目录选择

选择目录实现十分简单，只需要提供一个文件选择器即可，而且这个文件选择器只可以选择目录，当用户选择了对应的目录后，就可以将其选择的目录显示到 13.2.1 界面的 JTextField 中。文件选择器的代码如下。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\ConfigFrame.java

```
private JTextField field;
public FileChooser(JTextField field) {
    this.field = field;
    //设置只可以选择目录
    this.setSelectionMode(FileChooser.DIRECTORIES_ONLY);
}
//重写 JFileChooser 的方法
public void approveSelection() {
    //设置 JTextField 的值
    this.field.setText(this.getSelectedFile().getAbsolutePath());
    super.approveSelection();
}
```

用户选择目录后，就将其所选的目录的绝对路径显示到 JTextField 中，当点击确定的时候，就可以进行判断，以下代码为点击确定所执行的代码。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\ConfigFrame.java

```
//取得用户输入值
String mysqlHome = this.mysqlHomeField.getText();
//寻找用户选择的目录，判断是否可以找到 MySQL 安装目录下的 bin 目录
File file = new File(mysqlHome + MySQLUtil.MYSQL_HOME_BIN);
//找不到 MySQL 的安装目录，提示
```

```
if (!file.exists()) {  
    showMessage("请选择正确 MySQL 安装目录", "错误");  
    return;  
}
```

以上代码的黑体部分，需要去判断 MySQL 安装目录下的 bin 目录是否存在，如果没有该目录，则表示用户所选择的目录是错误的，弹出提示并返回。如果用户选择的目录是正确的话，就需要去读取管理器的配置文件。

### 13.3.2 读取和保存安装目录路径

用户选择了 MySQL 的安装目录后，我们需要将目录的绝对路径保存到一份配置文件中，这样的话，就可以不必每一次都去进行目录选择。提供一份 `mysql.properties` 的配置文件，以下为该配置文件的读取代码。

代码清单：code\mysql-manager\src\org\crazyit\mysql\util\FileUtil.java

```
//返回配置文件的 MYSQL_HOME 配置  
public static String getMySQLHome() {  
    File configFile = new File(MYSQL_PROPERTIES_FILE);  
    Properties props = getProperties(configFile);  
    return props.getProperty(MYSQL_HOME);  
}
```

以上代码中的 `MYSQL_PROPERTIES_FILE` 就是 `mysql.properties` 配置文件的相对路径，找到该文件后，就读取它的 `mysql.home` 属性。那么用户在进入 MySQL 安装目录选择界面的时候，就可以调用以上的方法去获得 MySQL 安装目录的值。

接下来实现保存安装目录的功能，在这之前，新建一个 `GlobalContext` 的类，用于保存管理器全局的一些信息，例如这里的 `mysql.home` 属性。以下代码实现保存配置的功能。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\ConfigFrame.java

```
//省略其他代码...  
//如果配置文件的值与用户输入的值不相等，则重新写入配置文件中  
if (!mysqlHome.equals(FileUtil.getMySQLHome())) {  
    FileUtil.saveMySQLHome(this.mysqlHomeField.getText());  
}  
GlobalContext ctx = new GlobalContext(mysqlHome);  
this.mainFrame = new MainFrame(ctx);  
this.mainFrame.setVisible(true);  
this.setVisible(false);
```

注意以上代码的判断，如果用户前一次所选择的 MySQL 安装目录与这一次所选择的目录不一致，则需要重新将新的目录信息保存到 `mysql.properties` 文件中。这些做的话，就不需要每一次进入系统都去修改配置文件。

### 13.3.3 读取连接信息

在得到 MySQL 安装目录，进入主界面时，还需要得到用户所有的连接信息，这些信息用来初始化主界面左边的树，管理器是针对 MySQL 数据库的，但是这些连接信息可以不记录到数据库，与保存 MySQL 安装目录一样，可以提供一些 `properties` 文件来保存，每一个连接作为一份 `properties` 文件。保存连接的信息我们在下面的章节中实现，这里主要实现读取的实现。

新建一个 `PropertiesHandler` 的接口，专门用于处理连接属性文件。该接口提供一个读取数据库连接配置文件的方法，并返回 `ServerConnection` 集合，`ServerConnection` 代表一个连接节点，并保存有一些数据库连接的信息，详细请看 13.2.2 中的 `ServerConnection` 类。以下代码读取一份 `properties`，

并返回一个 `Properties` 对象。

代码清单: `code\mysql-manager\src\org\crazyit\mysql\util\FileUtil.java`

```
//根据文件得到对应的 properties 文件
public static Properties getProperties(File propertyFile) throws IOException {
    Properties prop = new Properties();
    FileInputStream fis = new FileInputStream(propertyFile);
    prop.load(fis);
    fis.close();
    return prop;
}
```

那么在 `PropertiesHandler` 实现类中, 就可以读取相应目录下的所有 `properties` 文件。

代码清单: `code\mysql-manager\src\org\crazyit\mysql\system\PropertiesHandlerImpl.java`

```
//得到所有的连接信息
public List<ServerConnection> getServerConnections() {
    File[] propertyFiles = getPropertyFiles();
    List<ServerConnection> result = new ArrayList<ServerConnection>();
    for (File file : propertyFiles) {
        ServerConnection conn = createServerConnection(file);
        result.add(conn);
    }
    return result;
}

//将一份 properties 文件封装成 ServerConnection 对象
private ServerConnection createServerConnection(File file) {
    Properties prop = FileUtil.getProperties(file);
    ServerConnection conn = new ServerConnection(FileUtil.getFileName(file),
        prop.getProperty(FileUtil.USERNAME),
        prop.getProperty(FileUtil.PASSWORD),
        prop.getProperty(FileUtil.HOST),
        prop.getProperty(FileUtil.PORT));
    return conn;
}
```

得到所有的连接信息后, 先不需要初始化树, 需要将这些信息存放到一个对象中, 因为在下面的实现中, 这些类或者连接信息需要经常使用到。在 13.3.2 中提供了一个 `GlobalContext` 的类来表示管理器的上下文, 可以将这些连接信息放到该类中。

代码清单: `code\mysql-manager\src\org\crazyit\mysql\object\GlobalContext.java`

```
//存放所有服务器连接的集合
private Map<String, ServerConnection> connections = new HashMap<String, ServerConnection>();
//添加一个连接到 Map 中
public void addConnection(ServerConnection connection) {
    this.connections.put(connection.getConnectionName(), connection);
}
```

在 `GlobalContext` 中建立一个 `Map` 来保存这些连接信息, 并提供 `add` 方法, 由于这个 `Map` 是使用连接的名称作为 `key` 的, 所以就决定了在管理器中不允许出现重名的连接。那么在用户选择 `MySQL` 安装目录, 点击确定后, 就可以将连接加入到 `GlobalContext` 中, 用户点击确定按钮执行的部分代码。

代码清单: `code\mysql-manager\src\org\crazyit\mysql\ui\ConfigFrame.java`

```
//读取全部的服务器连接配置
List<ServerConnection> conns = ctx.getPropertiesHandler().getServerConnections();
for (ServerConnection conn : conns) ctx.addConnection(conn);
```

到此，MySQL 安装目录的功能已经实现，得到用户的各个连接信息后，就可以根据这些连接实现创建树的功能。

## 13.4 连接管理

进入主界面后，我们需要将各个连接信息创建一棵树，用户往后的各个操作，都与这些棵树息息相关。树的第一层节点是管理器中的各个连接，只需要得到各个连接后，以这些连接对象创建第一层节点即可。本小节将实现连接相关的功能，这些功能包括创建连接节点、打开连接、删除连接等。

### 13.4.1 创建连接节点

进入主界面时，我们已经可以得到 **GlobalContext** 对象，各个连接信息都保存在该对象中，因此可以根据这些连接信息来创建树。在创建树的时候，需要注意的是，我们只需要根据这些连接信息来创建第一层节点，而不需要再去创建下面的几层节点，当用户点击第一层节点（连接节点）的时候，再去访问该连接下面的数据库信息，正常得到这些数据库后，再创建数据库节点。

**MainFrame** 中创建连接节点的方法。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\MainFrame.java

```
//创建树中服务器连接的节点
private void createNodes(DefaultMutableTreeNode root) {
    Map<String, ServerConnection> conns = this.ctx.getConnections();
    for (String key : conns.keySet()) {
        ServerConnection conn = conns.get(key);
        //创建连接节点
        DefaultMutableTreeNode conntionNode = new DefaultMutableTreeNode(conn);
        root.add(conntionNode);
    }
}
```

**MainFrame** 中创建树的方法。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\MainFrame.java

```
//创建树
private void createTree() {
    DefaultMutableTreeNode root = new DefaultMutableTreeNode(new RootNode());
    //创建连接节点
    createNodes(root);
    this.treeModel = new DefaultTreeModel(root);
    //构造树
    JTree tree = new JTree(this.treeModel);
    //设置节点处理类
    TreeCellRenderer cr = new TreeCellRenderer();
    tree.setCellRenderer(cr);
    //设置监听器类
    tree.addMouseListener(new TreeListener(this));
    tree.setRootVisible(false);
    //添加右键菜单
    tree.add(this.treeMenu);
    this.tree = tree;
}
```

以上代码中的 `TreeCellRenderer` 为节点的处理类，具体的实现请看 13.2.2 中 `TreeCellRenderer` 类的实现。`TreeListener` 是一个鼠标事件监听器，当用户点击了树的连接节点后，需要建立连接，我们在下面的章节中实现。

### 13.4.2 打开连接

当用户点击连接节点后，就需要立即打开这个连接，我们使用了一个 `ServerConnection` 对象来保存连接的信息，并使用该对象来创建树中的连接节点，打开连接的时候，就可以根据连接的信息去尝试进行服务器连接，使用 `JDBC` 进行连接即可。如果成功进行连接，就马上创建该连接节点的子节点（数据库节点），如果不能成功连接，则弹出提示。`ServerConnection` 继承了 `ConnectionNode` 这个抽象类，`ConnectionNode` 中保存了一个 `JDBC` 的 `Connection` 对象，`ServerConnection` 中判断是否连接的标准是判断 `ConnectionNode` 的 `Connection` 对象是否为空，而且 `ServerConnection` 中需要实现父类（`ConnectionNode`）的 `connect` 方法，下面代码是 `ServerConnection` 对 `ConnectionNode` 的 `connect` 方法的实现。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\tree\ServerConnection.java

```
//实现父类的方法
public Connection connect() {
    //Connection 在本类中只有一个实例
    if (super.connection != null) return super.connection;
    Class.forName(DRIVER);
    Connection conn = createConnection("");
    super.connection = conn;
    return super.connection;
}
//创建连接，参数是数据库名称
public Connection createConnection(String database) throws Exception {
    Class.forName(DRIVER);
    Connection conn = DriverManager.getConnection(getConnectUrl() + database,
        this.username, this.password);
    return conn;
}
```

以上的代码中先判断 `ServerConnection` 的 `connection` 属性是否为空，如果该属性为空（没有连接）则进行创建。注意 `createConnection` 方法，该方法声明为 `public`，可以让外部去使用。下面实现树的节点监听器，当节点被选中后，就可以执行 `connect` 方法，但是需要注意的是，并不是每个节点都相同，只是连接节点被点击的时候才去进行连接。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\tree\TreeListener.java

```
public void mousePressed(MouseEvent e) {
    if (e.getModifiers() == MouseEvent.BUTTON1_MASK) {
        //左键点击，查看树的节点，调用 MainFrame 的打开节点方法
        this.mainFrame.viewTreeDatas();
    }
}
```

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\MainFrame.java

```
//点击树节点的操作
public void viewTreeDatas() {
    //获得选中的节点
    DefaultMutableTreeNode selectNode = getSelectNode();
    if (selectNode == null) return;
    //判断点击节点的类型
```



```

        if (selectNode.getUserObject() instanceof ServerConnection) {
            clickServerNode(selectNode); // 服务器连接节点
        }
    }
    // 点击服务器节点
    public void clickServerNode(DefaultMutableTreeNode selectNode) {
        // 暂时不实现
    }
}

```

连接节点被点击后，就会执行 `clickServerNode` 方法，该方法需要做的是先去验证被选中的节点是否可以连接，再创建该连接节点的子节点（数据库节点）。要创建数据库节点，就要得到该连接下面所有的数据库，执行 MySQL 的一句 `show databases` 就可以得到所有的数据库。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\tree\ServerConnection.java

```

// 获得一个服务器连接下面所有的数据库
public List<Database> getDatabases() {
    List<Database> result = new ArrayList<Database>();
    try {
        // 获得一个连接下面所有的数据库
        ResultSet rs = query("show databases");
        while (rs.next()) {
            String databaseName = rs.getString("Database");
            Database db = new Database(databaseName, this);
            result.add(db);
        }
        rs.close();
        return result;
    } catch (Exception e) {
        return result;
    }
}
// 查询并返回 ResultSet 对象
public ResultSet query(String sql) throws Exception {
    Statement stmt = getStatement();
    return stmt.executeQuery(sql);
}
}

```

使用 `Statement` 执行 `show databases` 就可以得到所有的数据库 `ResultSet` 对象，这里需要注意的是，我们需要得到 `Statement` 对象，直接使用 `ConnectionNode` 的 `connection` 属性去创建 `Statement` 对象即可，并不需要再去重新创建 JDBC 的 `Connection` 对象。查询到数据库的 `ResultSet` 对象后，就将 `Database` 列的值封装成一个 `Database` 对象，加入到结果集中即可。在本章中，一个 `Database` 对象代表一个数据库节点。那么现在就可以实现 `clickServerNode` 方法，点击了连接节点后，就会执行该方法。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\MainFrame.java

```

// 点击服务器节点
public void clickServerNode(DefaultMutableTreeNode selectNode) {
    ServerConnection server = (ServerConnection)selectNode.getUserObject();
    // 验证是否可以连接
    validateConnect(selectNode, server);
    // 创建服务器子节点
    buildServerChild(server, selectNode);
}

```

```

//创建数据库一层的节点（树的第二层）
public void buildServerChild(ServerConnection server,
    DefaultMutableTreeNode conntionNode) {
    //如果有子节点，则不再创建
    if (conntionNode.getChildCount() != 0) return;
    List<Database> databases = server.getDatabases();
    //再创建连接节点下面的数据节点
    for (Database database : databases) {
        DefaultMutableTreeNode databaseNode = new DefaultMutableTreeNode(database);
        //将数据库节点加入到连接节点中
        this.treeModel.insertNodeInto(databaseNode, conntionNode, conntionNode.getChildCount());
    }
}
//判断连接是否出错，适用于服务器节点和数据库节点
private void validateConnect(DefaultMutableTreeNode selectNode, ConnectionNode node) {
    //进行连接
    node.connect();
}

```

打开连接的功能已经实现，可以运行程序查看效果。总的来说，打开一个连接需要做的是：验证连接和创建数据库节点。

### 13.4.3 新建连接

在 13.2.4 中，我们已经提供了一个创建连接的界面，实现新建连接，只需要将用户输入的连接信息保存到一份 `properties` 文件中，再向树中添加一个连接节点即可。我们为接口 `PropertiesHandler` 添加一个 `saveServerConnection` 的方法，`PropertiesHandler` 是用于处理 `properties` 文件的接口，在 13.3.3 中已经创建。

`PropertiesHandler` 实现类对 `saveServerConnection` 的实现。

代码清单：code\mysql-manager\src\org\crazyit\mysql\system\PropertiesHandlerImpl.java

```

public void saveServerConnection(ServerConnection conn) {
    //得到配置文件名，这些 properties 文件存放于 connections 目录下
    String configFileName = FileUtil.CONNECTIONS_FOLDER +
        conn.getConnectionName() + ".properties";
    //创建 properties 文件
    File connConfigFile = new File(configFileName);
    //创建文件
    FileUtil.createNewFile(connConfigFile);
    Properties props = new Properties();
    props.setProperty(FileUtil.HOST, conn.getHost());
    props.setProperty(FileUtil.PORT, conn.getPort());
    props.setProperty(FileUtil.USERNAME, conn.getUsername());
    props.setProperty(FileUtil.PASSWORD, conn.getPassword());
    //将属性写入配置文件
    FileUtil.saveProperties(connConfigFile, props, "Connection " +
        conn.getConnectionName() + " config.");
}

```

`saveServerConnection` 实现简单，只需要将 `ServerConnection` 对象中的各个属性写到 `properties` 文件中即可。那么在 `ConnectionFrame`（新建连接界面）中，当用户输入各个信息点击确定后，就可以对这些连接进行保存，以下为点击确定执行的方法。

代码清单: code\mysql-manager\src\org\crazyit\mysql\ui\ConnectionFrame.java

```
//保存连接
private void saveConnection() {
    //得到用户输入的信息并返回一个 ServerConnection 对象
    ServerConnection conn = getDataConnectionFromView();
    //判断连接名称是否重复
    if (this.ctx.getConnection(conn.getConnectionName()) != null) {
        showMessage("已经存在相同名字的连接", "错误");
        return;
    }
    //直接保存, 不需要创建任何的连接, 添加到 GlobalContext 的连接 Map 中
    this.ctx.addConnection(conn);
    //保存到属性文件
    this.ctx.getPropertiesHandler().saveServerConnection(conn);
    this.mainFrame.addConnection(conn);
    this.setVisible(false);
}
```

注意以上代码的黑体部分, 需要进连接的名字进行判断, 先去 **GlobalContext** 的连接 **Map** 中获取 **ServerConnection** 对象, 如果能得到, 则表示已经存在相同名字的连接。保存到属性文件后, 就调用 **MainFrame** 的 **addConnection** 方法, 以下是 **addConnection** 方法的实现。

代码清单: code\mysql-manager\src\org\crazyit\mysql\ui\MainFrame.java

```
//在添加连接界面添加了一个连接后执行的方法, 向树中添加一个连接
public void addConnection(ServerConnection sc) {
    //得到要节点
    DefaultMutableTreeNode root = (DefaultMutableTreeNode)this.treeModel.getRoot();
    DefaultMutableTreeNode newChild = new DefaultMutableTreeNode(sc);
    //向要节点添加连接节点
    this.treeModel.insertNodeInto(newChild, root, root.getChildCount());
    if (root.getChildCount() == 1) this.tree.updateUI();
}
```

**addConnection** 方法直接使用 **DefaultTreeModel** 的 **insertNodeInto** 方法向树添加一个 **ServerConnection** 节点。运行程序并进行添加一个连接, 可以看到具体的效果。除了添加连接的功能外, 界面中还有一个测试连接的功能, 在添加连接前, 可以先测试一下服务器是否可以连接。以下是点击测试连接按钮触发的方法。

代码清单: code\mysql-manager\src\org\crazyit\mysql\ui\ConnectionFrame.java

```
//测试连接
private void checkConnection() {
    //从界面中得到连接信息
    ServerConnection conn = getDataConnectionFromView();
    try {
        conn.connect();
        showMessage("成功连接", "成功");
    } catch (Exception e) {
        showMessage(e.getMessage(), "警告");
    }
}
```

与打开连接一样, 都是使用 **ServerConnection** 的 **connect** 方法进行连接, 再捕获异常。保存的时候, 我们会再去从界面获取一个 **ServerConnection** 对象, 因此测试连接的 **ServerConnection** 对象与保存时候的 **ServerConnection** 是两个对象。

### 13.4.4 删除连接

用户选择了一个连接需要删除的时候，就需要提供一个删除连接的功能，删除连接的功能我们在右键菜单中提供，当用户选择了某个连接节点的时候，就弹出该菜单，该菜单已经在 13.2.9 中实现，下面实现删除连接的功能。首先我们需要明白的是，删除一个连接，就是从管理器中彻底删除这个连接信息，再从树中删除这个连接节点，最后还需要从 `GlobalContext` 的连接 `Map` 中删除该连接。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\MainFrame.java

```
//删除一个连接
private void removeConnection() {
    DefaultMutableTreeNode selectNode = getSelectNode();
    ServerConnection conn = (ServerConnection)selectNode.getUserObject();
    //从上下文件中删除
    this.ctx.removeConnection(conn);
    //从树节点中删除
    this.treeModel.removeNodeFromParent(selectNode);
}
```

当用户选择了某个连接节点的时候，选择右键菜单中的删除连接，就会触发上面的 `removeConnection` 方法，只需要为菜单对象添加 `ActionListener` 即可。先调用 `GlobalContext` 的删除连接方法将 `ServerConnection` 从全局上下文中删除，再使用 `DefaultTreeModel` 将该节点从树上删除。以下是 `GlobalContext` 中删除 `ServerConnection` 的方法（以上代码的黑体部分）。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\GlobalContext.java

```
//从 Map 中删除一个连接
public void removeConnection(ServerConnection connection) {
    //删除该连接的配置文件
    File configFile = new File(FileUtil.CONNECTIONS_FOLDER +
        connection.getConnectionName() + ".properties");
    configFile.delete();
    this.connections.remove(connection.getConnectionName());
}
```

`GlobalContext` 中的 `removeConnection` 方法，先删除 `properties` 文件，再从 `Map` 中删除该 `ServerConnection` 对象。

### 13.4.5 关闭连接

关闭一个服务器的连接，需要将 `ServerConnection` 对象的 `connection` 属性设置为 `true`，`connection` 属性保存在 `ServerConnection` 的父类 `ConnectionNode` 中，设置该属性为 `null` 后，连接节点的图标就会自然变成关闭的图标，因为 `ServerConnection` 中实现了 `ViewObject` 的 `getIcon` 方法。另外，还需要帮 `ServerConnection` 节点删除它的全部子节点。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\MainFrame.java

```
//删除一个节点的所有子节点
private void removeNodeChildren(DefaultMutableTreeNode node) {
    //获取节点数量
    int childCount = this.treeModel.getChildCount(node);
    for (int i = 0; i < childCount; i++) {
        //从最后一个开始删除
        this.treeModel.removeNodeFromParent((DefaultMutableTreeNode)node.getLastChild());
    }
}
```

```
//关闭服务器连接
private void closeConnection() {
    DefaultMutableTreeNode selectNode = getSelectNode();
    ServerConnection sc = (ServerConnection)selectNode.getUserObject();
    //将 ServerConnection 的连接对象设为 null
    sc.setConnection(null);
    //删除所有的子节点
    removeNodeChildren(selectNode);
    //设置树不选中
    this.tree.setSelectionPath(null);
}
```

以上代码的 `removeNodeChildren` 方法，从树中删除一个节点的所有子节点，用户选择了某个节点再进行删除节点后就会触发 `closeConnection` 方法。

## 13.5 数据库管理

数据库管理功能不多，包括打开数据库、关闭数据库和删除数据库，这三个功能与连接管理中的功能类似，例如打开连连与打开数据库，都需要创建子节点，但是每个数据库的子节点都只有三个，分别是表节点、视图节点和存储过程节点，而连接则是根据数据库来创建子节点的。在本章中，我们使用一个 `Database` 对象来代表一个数据库节点，具体请看 13.2.2 中的 `Database` 类。`Database` 对象与 `ServerConnection` 对象都是继承于 `ConnectionNode` 的，因此都需要去实现 `connect` 方法，并都有一个属性自己的 `Connection` 对象。实现打开数据库或者关闭数据库功能时，都与 `ServerConnection` 的实现类似。

### 13.5.1 打开数据库

当数据库节点被点击后，就可以进行打开操作，在 13.4.2 中，当树的某个节点被点击后，就会调用 `MainFrame` 的 `viewTreeDatas` 方法，在 13.4.2 中，我们只判断了用户点击服务器节点的情况，下面再帮该方法加入判断数据库节点的情况，当然，该方法还需要加入表节点、视图节点和存储过程节点的点击判断，判断用户点击的是哪种类型节点，再进行处理。

`MainFrame` 的 `viewTreeDatas` 方法。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\MainFrame.java

```
//判断点击节点的类型
if (selectNode.getUserObject() instanceof ServerConnection) {
    clickServerNode(selectNode); //服务器连接节点，在 13.4.2 中已经实现
} else if (selectNode.getUserObject() instanceof Database) {
    clickDatabaseNode(selectNode); //数据库连接节点
}
```

以上的代码判断了用户点击节点的类型，以下是上面代码中 `clickDatabaseNode` 的实现，点击数据库节点后，需要进行数据库连接，再为数据库节点添加三个子节点（表、视图和存储过程）。

`MainFrame` 的 `clickDatabaseNode` 方法。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\MainFrame.java

```
//创建数据库节点子节点
private void buildDatabaseChild(Database database,
    DefaultMutableTreeNode databaseNode) {
    //判断如果已经连接，则不创建节点
    if (databaseNode.getChildCount() != 0) return;
```

```

//创建三个子节点（表、视图、存储过程）
DefaultMutableTreeNode tableNode = new DefaultMutableTreeNode(new TableNode(database));
DefaultMutableTreeNode viewNode = new DefaultMutableTreeNode(new ViewNode(database));
ProcedureNode pNode = new ProcedureNode(database);
DefaultMutableTreeNode procedureNode = new DefaultMutableTreeNode(pNode);
//插入树中
this.treeModel.insertNodeInto(tableNode, databaseNode, databaseNode.getChildCount());
this.treeModel.insertNodeInto(viewNode, databaseNode, databaseNode.getChildCount());
this.treeModel.insertNodeInto(procedureNode, databaseNode, databaseNode.getChildCount());
}
//点击数据库节点
public void clickDatabaseNode(DefaultMutableTreeNode selectNode) {
    //获取点击树节点的对象
    Database database = (Database)selectNode.getUserObject();
    validateConnect(selectNode, database);
    //创建节点
    buildDatabaseChild(database, selectNode);
}

```

点击数据库节点与点击连接节点的实现类似，都是先进行验证连接，验证都是调用 `ConnectionNode` 的 `connect` 方法进行，而这个方法都由 `ServerConnection` 和 `Database` 分别进行实现。验证了连接后，再进行创建节点，数据库节点的子节点只有三个：表、视图和存储过程，以上代码的黑体部分创建这三个子节点。以下是 `Database` 对父类的 `connect` 方法的实现。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\tree\Database.java

```

//创建本类的连接对象
public Connection connect() {
    //如果已经连接， 则返回
    if (super.connection != null) return super.connection;
    //创建数据库连接
    super.connection = this.serverConnection.createConnection(this.databaseName);
    return super.connection;
}

```

我们在 13.2.2 中创建 `Database` 对象时，为该对象指定了一个构造器，构造 `Database` 对象必须要一个 `ServerConnection` 对象，表明一个 `Database` 所属的服务器连接，因为我们可以直接使用 `ServerConnection` 的 `createConnection` 方法去创建 `Connection` 连接，`createConnection` 方法在 13.4.2 中已经实现。

## 13.5.2 新建数据库

新建一个数据库，使用 JDBC 执行 `CREATE DATABASE` 即可实现，当用户选择了一个连接节点的时候，就可以选择弹出的右键菜单来创建数据库，如图 13.11 所示，接下来显示数据库创建界面，该界面只有一个 `JTextField`，给用户去输入数据库名称，在本章对应的是 `DatabaseFrame` 类。为 `Database` 对象新建一个 `create` 方法，该方法用于创建数据库。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\tree\Database.java

```

//创建数据库
public void create() {
    Statement stmt = this.serverConnection.getStatement();
    stmt.execute("create database " + this.databaseName);
}

```

为 `DatabaseFrame` 的确定按钮加入监听器并调用 `Database` 的 `create` 方法即可，需要注意的是，

显示 `DatabaseFrame` 的时候，需要将当前选择的连接节点对象（`ServerConnection`）也传递到 `DatabaseFrame` 中。创建了数据库后，以 `Database` 对象来创建一个树节点，添加到相应的连接节点下。

### 13.5.3 删除数据库

删除数据库，只需要使用 JDBC 执行 `DROP DATABASE` 语句即可实现，以下是删除数据库的实现。  
代码清单：code\mysql-manager\src\org\crazyit\mysql\object\tree\Database.java

```
//删除一个数据库
public void remove() {
    Statement stmt = this.serverConnection.getStatement();
    stmt.execute("drop database " + this.databaseName);
}
```

删除数据库后，还需要将该节点从树上删除。

代码清单：code\mysql-manager\src\org\crazyit\mysql\uiMainFrame.java

```
//删除一个数据库
private void removeDatabase() {
    //得到选择中的节点
    DefaultMutableTreeNode selectNode = getSelectNode();
    Database db = (Database)selectNode.getUserObject();
    db.remove();
    this.treeModel.removeNodeFromParent(selectNode);
}
```

### 13.5.4 关闭数据库

与 13.4.5 中关闭连接一样，都是将本类中的 `connection` 属性设置为 `null`，再将子节点全部删除。以下是关闭数据库的实现。

代码清单：code\mysql-manager\src\org\crazyit\mysql\uiMainFrame.java

```
//关闭数据库连接
private void closeDatabase() {
    DefaultMutableTreeNode selectNode = getSelectNode();
    Database db = (Database)selectNode.getUserObject();
    db.setConnection(null);
    //删除所有的子节点
    removeNodeChildren(selectNode);
    //设置树不选中
    this.tree.setSelectionPath(null);
}
```

以上代码的黑体部分已经在 13.4.5 中实现。到此，数据库的相关管理功能已经实现，数据库的功能相对比较简单，只需要使用 `CREATE DATABASE` 和 `DROP DATABASE` 即可实现，如果需要修改数据库，可以使用 `ALTER DATABASE` 实现。

## 13.6 视图管理

视图管理主要包括读取视图、新建视图、修改视图和查询视图。当用户选择了某个数据库，并点工具栏的视图菜单或者点击视图节点，就可以查询全部的视图，再选择某个具体的视图，点击鼠相当规模

右键，就弹出相关的右键菜单，具体的菜单在 13.2.10 中已经提供（图 13.14）。

### 13.6.1 读取视图列表

用户选择了某个数据库节点后，就可以打开这个数据库的连接，再点击这个数据库节点下面的视图节点，就可以查询这个数据库中所有的视图。在 13.4.2 中，当点击了树中的某个节点时，我们会执行一个 `viewTreeDatas` 方法，该方法在 13.5.1 中也实现了数据库节点的点击，现在再为该方法加入点击视图节点的实现。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\MainFrame.java

```
//点击树节点的操作
public void viewTreeDatas() {
    //获得选中的节点
    DefaultMutableTreeNode selectNode = getSelectNode();
    if (selectNode == null) return;
    //清空列表数据
    this.dataList.setListData(this.emptyData);
    //判断点击节点的类型
    if (selectNode.getUserObject() instanceof ServerConnection) {
        clickServerNode(selectNode); //服务器连接节点，在 13.4.2 中实现
    } else if (selectNode.getUserObject() instanceof Database) {
        clickDatabaseNode(selectNode); //数据库连接节点，在 13.5.1 中实现
    } else if (selectNode.getUserObject() instanceof ViewNode) {
        Database db = getDatabase(selectNode);
        clickViewNode(db); //视图节点
    }
}
```

在本章中，数据列表由一个 `JList` 实现，由于点击了视图节点会在 `JList` 中显示视图数据，因此我们需要将 `JList` 中原有的数据清空（以上代码的黑体部分）。当用户选择的是一个视图节点，那么就会执行 `clickViewNode` 方法（该方法在下面实现），该方法主要去读取数据库中的所有视图，再将数据放入 `JList` 中，我们将读取数据库视图的方法写在 `Database` 中。要查询所有的视图，我们需要到 MySQL 内置的数据库 `information_schema` 中的 `VIEWS` 表查询，该表保存了 MySQL 中所有的视图，因此查询的时候，我们需要加入数据库名称作为查询条件。

`Database` 中查询视图代码。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\tree\Database.java

```
//返回数据为中所有的视图
private ResultSet getViewsResultSet() throws Exception {
    Statement stmt = getStatement();
    //到 information_schema 数据库中的 VIEWS 表查询
    String sql = "SELECT * FROM information_schema.VIEWS sc WHERE " +
        "sc.TABLE_SCHEMA='" + this.databaseName + "'";
    ResultSet rs = stmt.executeQuery(sql);
    return rs;
}
```

以上代码执行一句查询的 SQL 并返回 `ResultSet` 对象，但是这样并不满足要求，我们需要将 `ResultSet` 对象转换成界面显示的数据格式。在列表中，我们使用一个 `ViewData` 代表一个视图，`ViewData` 对象在 13.2.2 中已经创建，该对象包含一个 `database` 和一个 `content`（`String` 类型）属性，`database` 属性代表这个 `ViewData` 对象所属的数据库，`content` 属性表示这个视图的内容，以下代码将 `ResultSet` 对象封装成 `ViewData` 集合。



代码清单: code\mysql-manager\src\org\crazyit\mysql\object\tree\Database.java

```
//返回这个数据库里的所有视图
public List<ViewData> getViews() {
    List<ViewData> result = new ArrayList<ViewData>();
    ResultSet rs = getViewsResultSet();
    while (rs.next()) {
        //得到视图的定义内容
        String content = rs.getString("VIEW_DEFINITION");
        ViewData td = new ViewData(this, content);
        //得到视图名称
        td.setName(rs.getString(TABLE_NAME));
        result.add(td);
    }
    rs.close();
    return result;
}
```

得到了视图对象的集合后,我们就可以实现点击视图节点的方法(本小节前面的 `clickViewNode` 方法),将得到的视图集合放入 `JList` 中,并创建相应的右键菜单(图 13.14)。

代码清单: code\mysql-manager\src\org\crazyit\mysql\uiMainFrame.java

```
//点击视图节点,查找全部的视图
private void clickViewNode(Database db) {
    List<ViewData> datas = db.getViews();
    this.dataList.setListData(datas.toArray());
    //显示视图后,创建右键菜单
    createViewMenu();
    //设置当前显示的数据类型为视图
    this.currentView = new ViewData(db, null);
}
```

注意最后还需要将当前的 `ViewObject` 设置为视图对象,用于标识当前所浏览的数据类型。实现效果如图 13.16 所示。

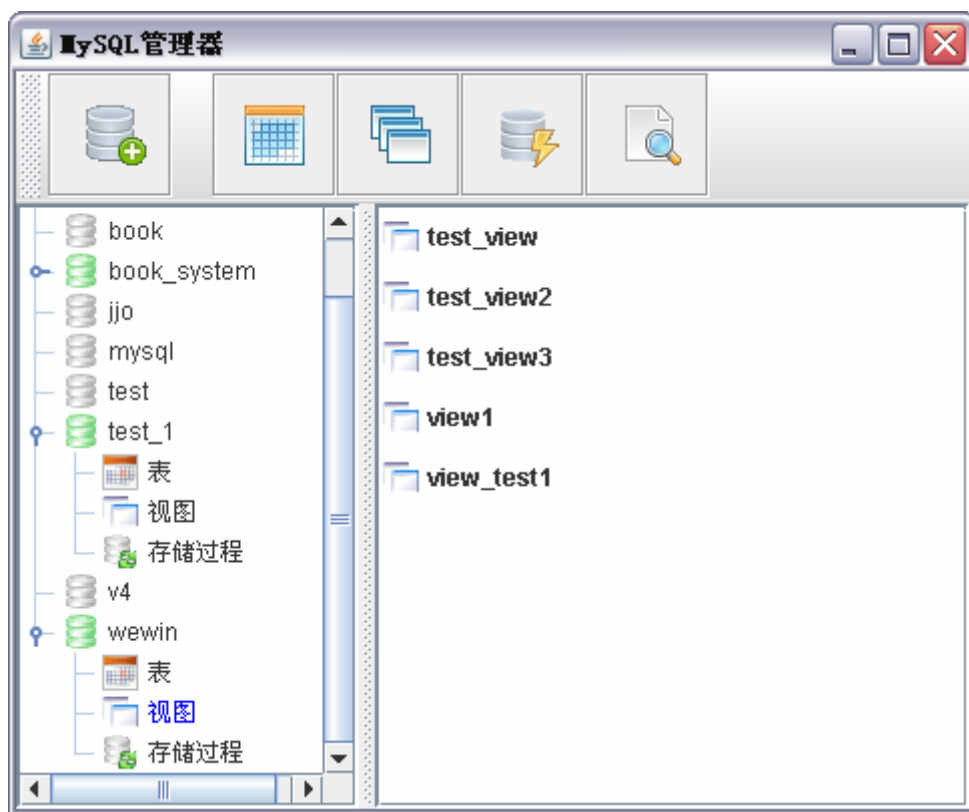


图 13.16 视图列表

### 13.6.2 新建视图

创建视图使用 JDBC 执行 **CREATE VIEW** 语句即可实现，视图界面只提供一个保存功能，由于我们创建视图与修改视图都是使用同一个界面，因此在执行保存的时候，就需要判断新增还是修改。当得到用户在视图界面输入的视图定义后，就可以执行 **CREATE VIEW** 语句进行创建视图。为 **ViewData** 对象加入一个创建视图的方法。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\list\ViewData.java

```
//创建视图
public void createView() {
    //拼装 CREATE VIEW 语句
    String sql = MySQLUtil.CREATE_VIEW + name + " " +
        MySQLUtil.AS + " " + content;
    database.getStatement().execute(sql);
}
```

用户点击保存，弹出另外一个窗口让用户输入视图名称，最后调用上面的 **createView** 方法，即可以创建视图，

### 13.6.3 修改视图与删除视图

与创建视图一样，使用同样的界面，只是执行不同的 SQL 语句，修改视图可以使用 **ALTER VIEW** 即可，为 **ViewData** 对象加入修改视图的方法。当修改完视图后，调用修改方法即可。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\list\ViewData.java

```
//修改视图
```

```
public void alterView() {
    String sql = MySQLUtil.ALTER_VIEW + name + " " + MySQLUtil.AS
        + " " + content;
    database.getStatement().execute(sql);
}
```

同样地，删除视图使用 JDBC 执行 DROP VIEW 即可实现。

ViewData:

```
//删除视图
public void dropView() {
    String sql = MySQLUtil.DROP_VIEW + this.name;
    database.getStatement().execute(sql);
}
```

到此，查询全部视图、创建视图、修改视图和删除视图功能已经实现，但是，还缺少一个最重要的功能，就是查看视图。当我们选择了某个视图进行双击操作的时候，就需要浏览该视图的数据，这一个功能我们将在下面的章节中实现。

## 13.7 存储过程与函数管理

存储过程与函数管理使用相同的界面，因此我们可以一起实现，它们的区别在于是否有返回值，通过一些界面判断即可实现。与视图管理一样，都是有新增、修改和删除功能。

### 13.7.1 新增存储过程和函数

存储过程和函数的界面已经在 13.2.7 中创建，得到存储过程或者函数的定义、参数、返回值（函数）与名称后，就可以使用命令去创建存储过程或者函数。创建存储过程使用 **CREATE PROCEDURE**，创建函数使用 **CREATE FUNCTION**。在本章中，视图数据使用的是一个 ViewData 对象（13.6 章节），在 13.2.2 中也创建了一个 ProcedureData 对象来表示一个存储过程的数据对象，因此将创建存储过程或者函数加入到该类中即可。

ProcedureData 创建存储过程方法。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\list\ProcedureData.java

```
//创建存储过程
public void createProcedure() {
    String sql = MySQLUtil.CREATE_PROCEDURE + this.name +
        "(" + this.arg + ")" + this.content;
    this.database.getStatement().execute(sql);
}
```

ProcedureData 中创建函数方法。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\list\ProcedureData.java

```
//创建函数
public void createFunction() {
    String sql = MySQLUtil.CREATE_FUNCTION + this.name +
        "(" + this.arg + ") returns " + this.returnString + " " + this.content;
    this.database.getStatement().execute(sql);
}
```

存储过程对象（ProcedureData）与视图对象（ViewData）一样，都是属于某个数据库的，因此这两个对象都会保存一个数据库的属性，直接就可以通过数据库对象（Database）的 **getStatement** 方法得到 **Statement** 对象，再执行 SQL 语句。

### 13.7.2 修改存储过程与函数

修改存储过程（函数）与新增存储过程（函数）使用的是相同的界面，因此在保存的时候需要作出判断。与修改视图不同的是，修改存储过程或者函数，不使用 `ALTER PROCEDURE`（`ALTER FUNCTION`）来实现，这是由于 MySQL 中的 `ALTER PROCEDURE` 和 `ALTER FUNCTION` 并不能修改存储过程或者函数的方法体与参数，因此，实现时需要将原来的存储过程或者函数先删除，再重新创建。

ProcedureData 中修改存储过程。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\list\ProcedureData.java

```
//修改存储过程
public void updateProcedure() {
    //修改存储过程需要先把原来的先删除
    //删除语句
    String dropSQL = MySQLUtil.DROP_PROCEDURE + this.name;
    this.database.getStatement().execute(dropSQL);
    //创建语句
    String createSQL = MySQLUtil.CREATE_PROCEDURE + this.name +
        "(" + this.arg + ")" + this.content;
    this.database.getStatement().execute(createSQL);
}
```

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\list\ProcedureData.java

```
//修改函数
public void updateFunction() {
    //修改需要先把原来的先删除
    String dropSQL = MySQLUtil.DROP_FUNCTION + this.name;
    this.database.getStatement().execute(dropSQL);
    String createSQL = MySQLUtil.CREATE_FUNCTION + this.name +
        "(" + this.arg + ") returns " + this.returnString + "\n" + this.content;
    this.database.getStatement().execute(createSQL);
}
```

以上两个方法就可以修改存储过程和函数，但是如果一旦存储过程或者函数编写有误，那么就会将原来的存储过程或者函数删除，为了解决这个问题，可以将原来的存储过程改名，再创建一个修改后的存储过程，如果创建失败，就将改名后的旧的存储过程改回来，这样就可以确保错误发生后无法恢复原来的存储过程。修改存储过程或者函数的名称使用 `ALTER PROCEDURE` 或者 `ALTER FUNCTION` 即可实现。

在修改存储过程与函数的时候，我们就使用了 `DROP PROCEDURE` 和 `DROP FUNCTION` 来删除一个存储过程和函数，删除存储过程和函数不再详细描述。存储过程或者函数的调用可以使用 `CALL` 来调用，在实现了 SQL 查询功能后，就可以执行一句 `CALL` 的 SQL 来调用查看效果。

## 13.8 表管理

表管理在本章中相对较难，我们需要从界面中得到创建表的信息，例如字段信息、外键字段信息等。在修改表的时候，用户在界面的表格中会进行各种操作，操作完后进行保存，就需要收集这些被操作过的数据，再进表的修改。

在 13.2.5 中，我们已经创建的表管理界面（如 13.2.5 中的图 13.7 所示），对应的是 `TableFrame` 类，界面中存的有两个列表对象，分别是字段列表和外键字段列表，对应的类是 `FieldTable` 与 `ForeignTable`，它们都继承于 `JTable`。字段列表有以下操作：

- ☐ 新字段：向字段列表的尾部追加一个新的字段行。
- ☐ 插入字段：在所选择的行前面插入一个字段行。
- ☐ 删除字段：删除所选的行。
- ☐ 设置默认值：在文本框中输入该字段的默认值。
- ☐ 设置自动增长：设置字段是否可以自动增长。

外键字段有以下操作：

- ☐ 新外键：向表外尾部追加一个新的外键。
- ☐ 删除外键：删除一个选择的外键。

### 13.8.1 新字段

为了能体现一个字段，我们新建一个字段对象 **Field**，该对象保存一个字段的的所有信息，包括名称，长度等一系列的字段信息。

代码清单：code\mysql-manager\src\org\crazyit\mysql\table\object\Field.java

```
private String fieldName; //字段名
private String type; //字段类型
private boolean allowNull = true; //允许空，默认允许为空
private boolean isPrimaryKey = false; //是否主键，是主键为 true，否则为 false
private String defaultValue; //默认值
private boolean autoIncrement = false; //是否自动增长
private TableData table; //该字段所属的表
private String uuid; //标识这个字段的 uuid
//省略 setter 和 getter 方法
```

接下来，在 **TableFrame**（表管理界面）中，创建一个字段集合用来保存当前界面所显示的字段，那么如果进行新建字段操作，就可以对该集合进行操作了。下面实现刷新字段列表的方法，由于在加入新字段、修改字段或者删除字段后，都需要将列表进行一次刷新。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\TableFrame.java

```
//刷新字段列表
public void refreshFieldTable() {
    DefaultTableModel tableModel = (DefaultTableModel)this.fieldTable.getModel();
    //设置数据
    tableModel.setDataVector(getFieldDatas(), this.fieldTable.getFieldTableColumn());
    //设置列表样式，包括行高、列宽等
    this.fieldTable.setTableFace();
}
//得到字段列表数据
public Vector getFieldDatas() {
    Vector datas = new Vector();
    for (int i = 0; i < this.fields.size(); i++) {
        Field field = this.fields.get(i);
        Vector data = new Vector();
        data.add(field.getFieldName()); //字段名称
        data.add(field.getType()); //字段类型
        data.add(getNullIcon(field)); //获得是否允许空的图片
        data.add(getPrimaryKeyIcon(field)); //获得主键图片
        datas.add(data);
    }
    return datas;
}
```

```
}
```

以上代码中的黑体部分，就是当前界面中的字段集合。在新建一个字段后，就可以使用 `refreshFieldTable` 方法对字段列表进行刷新。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\TableFrame.java

```
//加入新字段
private void newField() {
    Field field = new Field();
    this.fields.add(field);
    //刷新字段列表
    refreshFieldTable();
    //如果是修改状态，则添加 addFields 集合中
    if (this.table.getName() != null) this.addFields.add(field);
}
```

以上的黑体代码，是在用户是行修改表的时候才需要，我们创建一个 `addFields` 集合，用来保存用户添加过的字段（修改的时候）。该集合的作用，我们将修改表的时候详细描述。

### 13.8.2 插入字段与删除字段

插入新字段，只需要得到用户当前所选择的行，并在该行的前面加入一个数据行即可，需要注意的是，当用户没有选择任意一行的时候，就可以调用新字段的方法，即 13.8.1 中创建新字段的方法。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\TableFrame.java

```
//插入新字段
private void insertField() {
    //得到选择中的行索引
    int selectRow = this.fieldTable.getSelectedRow();
    if (selectRow == -1) {
        //没有选中，调用加新字段方法，加入新字段
        newField();
        return;
    }
    Field field = new Field();
    this.fields.add(selectRow, field);
    //刷新字段列表
    refreshFieldTable();
    //如果是修改状态，则添加 addFields 集合中
    if (this.table.getName() != null) this.addFields.add(field);
}
```

删除字段实现与插入字段一样，只需要将字段从集合中删除并刷新列表即可，另外，如果是修改表的话，就需要加入另外的操作，在下面修改表的章节中描述。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\TableFrame.java

```
//删除字段
private void deleteField() {
    //得到选中的行
    int selectRow = this.fieldTable.getSelectedRow();
    if (selectRow == -1) return;
    //得到用户所选择的 Field 对象
    Field field = this.fields.get(selectRow);
    if (field == null) return;
    //从字段集合中删除
```

```

        this.fields.remove(field);
        //刷新列表
        refreshFieldTable();
    }

```

### 13.8.3 编辑字段

当用户在字段列表中对字段的信息进行修改时，就需要得到相应的 **Field** 对象，并设置新的信息。当用户对列表停止编辑的时候，就可以触发相应的方法。这里需要注意的是，当停止编辑的时候，需要修改对应的 **Field** 对象，只需要修改该对象的字段名称与字段类型，因为这两个属性才可以输入，其他两个属性（是否允许空和主键）进行选择才会发生值的改变。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\table\FieldTable.java

```

//重写 JTable 的方法，列表停止编辑的时候触发该方法
public void editingStopped(ChangeEvent e) {
    int column = this.getEditingColumn();
    int row = this.getEditingRow();
    super.editingStopped(e);
    //获得当前编辑的列名
    DefaultTableModel model = (DefaultTableModel)this.getModel();
    String columnName = model.getColumnName(column);
    //得到编辑后的单元格的值
    String value = (String)this.getValueAt(row, column);
    if (columnName.equals(FIELD_NAME)) {
        //更改字段名称
        this.tableFrame.changeFieldName(row, value);
    } else if (columnName.equals(FIELD_TYPE)) {
        //更改字段类型
        this.tableFrame.changeFieldType(row, value);
    }
}

```

在列表停止编辑的时候，得到用户所编辑的单元格的行索引和编辑后的值，再调用 **TableFrame** 的方法进行修改字段名和字段类型。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\TableFrame.java

```

//字段列表的字段名称，同步去修改字段集合中的字段名称值
public void changeFieldName(int row, String value) {
    //得到相应的 Field 对象
    Field field = this.fields.get(row);
    if (field == null) return;
    field.setFieldName(value);
}

//字段列表的字段类型，同步去修改字段集合中的字段类型值
public void changeFieldType(int row, String value) {
    //得到相应的 Field 对象
    Field field = this.fields.get(row);
    if (field == null) return;
    field.setType(value);
}

```

那么用户对列表进行修改后，就可以同步的去修改 **TableFrame** 中的相应对象的字段名和字段类型。另外，如果用户点击了“允许空”和“主键”列，就需要同步去修改集合中的 **Field** 对象。修改 **FieldTable**

的 `selectCell` 方法，该方法在 13.2.5 中已经实现了部分，当用户选择了列表的时候，就会触发该方法。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\table\FieldTable.java

```
//鼠标在 JTable 中点击的时候触发该方法
private void selectCell() {
    int column = this.getSelectedColumn();
    int row = this.getSelectedRow();
    if (column == -1 || row == -1) return;
    //修改图片列
    selectAllowNullColumn(row, column);
    selectPrimaryKeyColumn(row, column);
    //设置点击后会改变的值，注意是点击，并不是输入，因此只会更改允许空和主键
    changeClickValue(row, column);
}
```

以上的黑体代码为新加的代码。其中的 `changeClickValue` 为改变“允许空”与“主键”这两列的值，当用户点击了这两列的时候，就需要同步修改 `TableFrame` 的 `fields` 集合。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\table\FieldTable.java

```
//当发生鼠标点击单元格事件的时候，改变值，一般只改变允许空和主键列
private void changeClickValue(int row, int column) {
    //得到主键列
    TableColumn primaryColumn = this.getColumn(PRIMARY_KEY);
    if (primaryColumn.getModelIndex() == column) {
        this.tableFrame.changePrimaryKeyValue(row);
    }
    //得到允许空列
    TableColumn allowNullColumn = this.getColumn(ALLOW_NULL);
    if (allowNullColumn.getModelIndex() == column) {
        this.tableFrame.changeAllowNullValue(row);
    }
}
```

判断用户所点击的单元格所属的列，如果是“允许空”或者“主键”列，就可以调用 `TableFrame` 中的方法去修对应 `Field` 对象的属性值。

### 13.8.4 设置默认值与自动增长

界面中提供了一个文本框，可以设置某个字段的默认值，提供了一个多选框，可以设置字段是否可以自动增长。我们可以为文本框加入按键事件，当用户在文本框中进行输入时，就可以改变该字段的默认值。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\TableFrame.java

```
//改变字段的默认值
public void changeDefaultValue() {
    //得到选中的行
    int selectRow = this.fieldTable.getSelectedRow();
    if (selectRow == -1) return;
    //取得默认值
    String defaultValue = this.defaultField.getText();
    //取得当前编辑的 Field 对象
    Field field = this.fields.get(selectRow);
    //设置字段默认值
    field.setDefaultValue(defaultValue);
}
```



```
}
```

同样地，与设置默认值一样，也可以为多选框加入监听器，如果发生点击事件时，就执行某个方法。  
代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\TableFrame.java

```
//点击自动增长 checkBox 的方法
private void clickIsAutoIncrementBox() {
    //得到字段列表中所选中的行索引
    int row = this.fieldTable.getSelectedRow();
    if (row == -1) return;
    //得到当前所选择了 Field 对象
    Field field = this.fields.get(row);
    //设置 Field 对象中的自动增长属性
    if (this.isAutoIncrementBox.isSelected()) field.setAutoIncrement(true);
    else field.setAutoIncrement(false);
}
```

那么用户在设置默认值或者自动增长的时候，就可以同步将默认值与自动增长的标识保存到 TableFrame 的 fields 集合中。但是，还需要去修改点击字段列表的方法，将默认值与自动增长的值展现到界面的元素中。修改 FieldTable 的 selectCell 方法，列表被点击时，会触发该方法。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\table\FieldTable.java

```
//鼠标在 JTable 中点击的时候触发该方法
private void selectCell() {
    //省略其他代码
    //修改默认值
    this.tableFrame.setDefaultValue(row);
    //修改是否自动增长的 checkbox
    this.tableFrame.setIsAutoIncrement(row);
}
```

点击了列表某行的时候，就可以得到相应的 Field 对象，再设置界面的文本框和多选框即可。

### 13.8.5 新外键

与实现新字段一样，新建一个 ForeignField 对象来代表一个外键，并在 TableFrame 中创建一个集合来保存当前界面中的外键。

代码清单：code\mysql-manager\src\org\crazyit\mysql\table\object\ForeignField.java

```
private String constraintName; //约束的名称
private Field field; //被约束的字段，根据该字段可以找出该外键对象所属于的表
private Field referenceField; //外键的字段，可以根据此属性找出该关系中的外键表
private String onDelete; //级联删除策略
private String onUpdate; //级联更新策略
private String referenceTableName; //约束表的名称
private String referenceFieldName; //约束字段的名称
private String uuid; //字段的 uuid
//省略 setter 和 getter 方法
```

在 13.2.5 中已经创建了外键列表对象 (ForeignTable)，新建刷新外键列表的方法。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\TableFrame.java

```
//刷新外键字段列表
public void refreshForeignFieldTable() {
    //设置外键数据
    DefaultTableModel tableModel = (DefaultTableModel)this.foreignTable.getModel();
    tableModel.setDataVector(getForeignDatas(), this.foreignTable.getForeignColumns());
}
```

```
//设置外键列表的样式
this.foreignTable.setTableFace();
}
```

加入一个外键的时候，就可以调用 `refreshForeignFieldTable` 方法刷新外键列表。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\TableFrame.java

```
//新增一个外键字段
private void newForeignField() {
    ForeignField foreignField = new ForeignField();
    this.foreignFields.add(foreignField);
    //设置该外键的 constraintName，用 UUID 设置
    foreignField.setConstraintName(UUID.randomUUID().toString());
    //刷新外键列表
    refreshForeignFieldTable();
    //如果是修改状态，加到添加的外键集合中
    if (this.table.getName() != null) this.addForeignFields.add(foreignField);
}
```

以上的黑体代码，与 13.8.1 中新字段一样，都是在修改表的时候需要使用到的代码，将在下面修改表的章节中加以描述。这里需要注意的是，为了标识新加的“外键”在数据库中的唯一性，因为需要将 `ForeignField` 的 `constraintName` 属性设置为唯一的。

### 13.8.6 删除一个外键

删除外键实现比较简单，只需要从 `TableFrame` 的“外键”集合中删除即可。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\TableFrame.java

```
//删除一个字段
private void deleteForeignField() {
    //得到选中的行
    int selectRow = this.foreignTable.getSelectedRow();
    if (selectRow == -1) return;
    //得到选中的外键对象
    ForeignField field = this.foreignFields.get(selectRow);
    if (field == null) return;
    //从字段集合中删除
    this.foreignFields.remove(field);
}
```

前面几个小节中，我都讲解了如何实现表管理中的一些界面操作，接下来实现具体的表管理，包括查询表信息、保存表和修改表。

### 13.8.7 查询字段信息

查询一个表的信息需要到 MySQL 的系统表中查询，这些信息包括字段信息与外键信息等。由于我们在 `TableFrame` 中建立了一个字段集合来保存当前界面中的字段信息，因此，只需要从数据库中查询所有的表字段并封装成一个 `Field` 集合即可。在本章中，一个表由一个 `TableData` 对象来代表，`TableData` 中包含了一个 `Database` 对象，`Database` 对象可以取到数据库的连接信息，可以将查询字段的方法写到 `TableData` 中。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\list\TableData.java

```
//获得查询字段的 SQL
private String getFieldSQL() {
```

```

StringBuffer sql = new StringBuffer();
sql.append("SELECT * FROM information_schema.COLUMNS sc")
.append(" WHERE sc.TABLE_SCHEMA=")
.append(this.database.getDatabaseName() + " ")
.append(" AND sc.TABLE_NAME=")
.append(this.name + " ")
.append("ORDER BY sc.ORDINAL_POSITION");
return sql.toString();
}

```

getFieldSQL 方法是返回一句查询的 SQL，到系统表中查询一个表的所有字段，使用 JDBC 执行 getFieldSQL 方法返回的 SQL，就可以得到 ResultSet 对象，再得到各个列的值。

我们需要从查询到的 ResultSet 中得到以下值：

- ☐ COLUMN\_NAME：字段名。
- ☐ COLUMN\_TYPE：字段类型。
- ☐ IS\_NULLABLE：是否允许空。
- ☐ COLUMN\_KEY：如果是主键，那么该值为“PRI”。
- ☐ COLUMN\_DEFAULT：该字段的默认值。
- ☐ EXTRA：如果该值为 auto\_increment，则表示是自动增长。

得到以上的值，就可以封装成一个字段对象，并加到结果集合中，那么 TableFrame 就可以根据这个集合来显示字段数据。

### 13.8.8 查询外键信息

查询外键信息与查询字段信息一样，都是到 MySQL 的系统表（KEY\_COLUMN\_USAGE）中进行查询，但是，如果使用的 MySQL5.0，则不能到系统表中查询到一个外键的 ON DELETE 和 ON UPDATE 的值。如果使用的是 MySQL5.1，就可以到系统表中查询到一个字段的这两个值。本章使用的 MySQL 版本是 5.0，因此要得到 ON DELETE 和 ON UPDATE 的值，就需要得到建表时的 SQL，并对该句 SQL 进行分析，得到外键的 ON DELETE 和 ON UPDATE，这就是本章开头所讲的 MySQL5.0 与 MySQL5.1 的差别对我们开发这个管理器所产生的影响。

执行下面的 SQL 就可以返回外键字段信息的 ResultSet：

```

SELECT * FROM information_schema.KEY_COLUMN_USAGE sc WHERE sc.TABLE_SCHEMA='数据库名'
AND sc.TABLE_NAME='表名' AND sc.REFERENCED_COLUMN_NAME <> "" ORDER BY sc.COLUMN_NAME

```

得到的 ResultSet 里面包含有如下字段：

- ☐ COLUMN\_NAME：外键列名。
- ☐ CONSTRAINT\_NAME：约束的名称。
- ☐ REFERENCED\_TABLE\_NAME：约束的表名。
- ☐ REFERENCED\_COLUMN\_NAME：约束的字段名。

得到约束的字段名后，就可以再次到系统表（COLUMNS）中查询约束的字段。得到这些信息后，就可以创建一个 ForeignField 对象，但是，ForeignField 中还包含有 onDelete 和 onUpdate 两个属性，为了得到这个属性，我们需要得到创建表的 SQL，并对该句 SQL 进行分析。如果你使用的是 MySQL5.1，就可以直接到系统表中查询。得到创建表的 SQL，可以使用 JDBC 执行 SHOW CREATE TABLE 来实现。

以下方法分析创建表的 SQL，并得到 ON DELETE 和 ON UPDATE 信息。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\list\TableData.java

```

//返回 ON DELETE 或者 ON UPDATE 的值
private String getOnValue(String createSQL, ForeignField foreignField,

```

```

        String on) {
            String constraintName = foreignField.getConstraintName();
            //以逗号将其分隔
            String[] temp = createSQL.split(",");
            for (int i = 0; i < temp.length; i++) {
                String tempString = temp[i];
                //如果遇到外键的字符串, 则进行处理
                if (tempString.indexOf("CONSTRAINT `" + constraintName + "`") != -1) {
                    //如果遇到 ON DELETE 或者 ON UPDATE, 则进行处理, 返回 ON DELETE 或者 ON UPDATE
                    的值
                    if (tempString.indexOf(on) != -1) {
                        //得到 ON DELETE 或者 ON UPDAT 的位置
                        int onIndex = tempString.indexOf(on) + on.length() + 1;
                        String value = tempString.substring(onIndex, onIndex + 7);
                        if (value.indexOf("NO ACTI") != -1) return "NO ACTION";
                        else if (value.indexOf("RESTRIC") != -1) return "RESTRICT";
                        else if (value.indexOf("CASCADE") != -1) return "CASCADE";
                        else if (value.indexOf("SET NUL") != -1) return "SET NULL";
                    }
                }
            }
            return null;
        }
    }
}

```

得到各个信息并封装成 **ForeignField** 的集合后, 就可以将这个集合放到 **TableFrame** 中。**TableFrame** 中保存了一个外键字段的集合, 在 13.8.5 新外键中创建该集合。得到字段与外键字段的集合后, 就可以在 **TableFrame** 中初始化列表, 使用 **DefaultTableModel** 的 **setDataVector** 方法即可初始列表数据。

### 13.8.9 新建表

我们可以 **TableFrame** (表管理界面) 中得到字段的集合, 集合中保存了一系列的 **Field** 对象, 那么我们在创建表的时候, 就可以根据这些 **Field** 对象的属性来拼装 **CREATE TABLE** 的 SQL 语句, 然后使用 **JDBC** 执行即可。

编写拼装 SQL 的程序, 最终得出的 SQL 语句如下:

```

CREATE TABLE IF NOT EXISTS `table3` (`field1` int(10)AUTO_INCREMENT, `field2` varchar(10)DEFAULT
'test', `field3` int(10), FOREIGN KEY (`field3`) REFERENCES `table_2` (`field_3`) ON DELETE CASCADE ON
UPDATE CASCADE, PRIMARY KEY(`field1`))

```

具体实现在 **TableData** 中的 **addTable** 方法, 拼装 SQL 语句的方法是 **TableData** 中的 **getTableSQL** 方法, **getTableSQL** 方法先拼装 **CREATE TABLE** 命令, 再依次拼装字段的 SQL、外键字段的 SQL 和创建主键的 SQL。

拼装字段 SQL 的方法。

代码清单: code\mysql-manager\src\org\crazyit\mysql\object\list\TableData.java

```

//根据字段创建 SQL
private void createField(StringBuffer sql, List<Field> fields) {
    for (Field f : fields) {
        sql.append("`" + f.getFieldName() + "` ")
        .append(f.getType());
        //自增长加入 AUTO_INCREMENT
        if (f.isAutoIncrement()) sql.append(AUTO_INCREMENT + " ");
        //该字段不允许为空, 加入 NOT NULL
    }
}

```

```

        if (!f.isAllowNull()) sql.append(NOT_NULL + " ");
        //该字段有默认值,并且不是自动增长
        if (!f.isAutoIncrement()) {
            if (f.getDefaultValue() != null) sql.append(DEFAULT + " " + f.getDefaultValue() + " ");
        }
        sql.append(",");
    }
}

```

拼装外键字段的 SQL。

代码清单: code\mysql-manager\src\org\crazyit\mysql\object\list\TableData.java

```

//创建外键 SQL
private void createForeignFields(StringBuffer sql,
    List<ForeignField> foreignFields) {
    for (ForeignField f : foreignFields) {
        sql.append(FOREIGN_KEY)
            .append(" (" + f.getField().getFieldName() + ") ")
            .append(REFERENCES)
            .append("`" + f.getReferenceField().getTable().getName() + "` ")
            .append("(" + f.getReferenceField().getFieldName() + ") ");
        if (f.getOnDelete() != null) {
            sql.append(ON_DELETE + " " + f.getOnDelete() + " ");
        }
        if (f.getOnUpdate() != null) {
            sql.append(ON_UPDATE + " " + f.getOnUpdate() + " ");
        }
        sql.append(",");
    }
}

```

拼装主键的 SQL。

代码清单: code\mysql-manager\src\org\crazyit\mysql\object\list\TableData.java

```

//创建主键 SQL
private void createPrimary(StringBuffer sql, List<Field> fields) {
    for (Field f : fields) {
        if (f.isPrimaryKey()) {
            sql.append(PRIMARY_KEY).append("(" + f.getFieldName() + ")").append(",");
        }
    }
}

```

得到创建表的 SQL 后, 使用 JDBC 执行即可。如果创建失败, 可以将异常信息显示到界面中。

### 13.8.10 修改表

实现修改表的功能较为复杂, 由于保存在界面中可能涉及的操作包括添加字段、修改字段、删除字段、添加外键字段、修改外键字段和删除外键字段, 因此修改表的话, 我们需要知道用户添加、修改、删除了哪些字段与外键字段。也就是说, 用户每次进行添加、修改、删除字段与外键字段的时候, 我们都需要对用户所操作的数据进行保存, 最后, 得到这些数据后, 就可以使用 **ALTER TABLE** 来进行表的修改。在 13.8.1 新字段中, 创建一个字段的时候, 另外建立了一个集合来保存用户所添加的字段, 因此还需要在修改和删除操作的时候, 保存用户所操作的数据。

得到用户操作过的数据集后, 我们就可以为 **TableData** 加入一个修改表的方法, 将这些集合里面

的对象都转换成 SQL，然后使用 JDBC 执行。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\list\TableData.java

```
/**
 * 修改一个表
 * @param addFields 需要添加的字段
 * @param updateFields 修改的字段
 * @param dropFields 删除的字段
 * @param addFF 添加的外键
 * @param updateFF 修改的外键
 * @param dropFF 删除的外键
 */
public void updateTable(List<Field> addFields, List<UpdateField> updateFields,
    List<Field> dropFields, List<ForeignField> addFF,
    List<UpdateForeignField> updateFF, List<ForeignField> dropFF) {
    //得到添加字段的 SQL
    List<String> addFieldSQL = getAlterAddFieldSQL(addFields);
    //得到修改字段的 SQL
    List<String> updateFieldSQL = getAlterUpdateFieldSQL(updateFields);
    //得到删除字段的 SQL
    List<String> dropFieldSQL = getAlterDropFieldSQL(dropFields);
    //得到添加外键的 SQL
    List<String> addFFSQL = getAlterAddForeignFieldSQL(addFF);
    //得到修改外键的 SQL
    List<String> updateFFSQL = getAlterUpdateForeignFieldSQL(updateFF);
    //得到删除外键的 SQL
    List<String> dropFFSQL = getAlterDropForeignFieldSQL(dropFF);
    try {
        Statement stmt = database.getStatement();
        for (String s : addFieldSQL) stmt.addBatch(s);
        for (String s : updateFieldSQL) stmt.addBatch(s);
        for (String s : dropFieldSQL) stmt.addBatch(s);
        for (String s : addFFSQL) stmt.addBatch(s);
        for (String s : updateFFSQL) stmt.addBatch(s);
        for (String s : dropFFSQL) stmt.addBatch(s);
        stmt.executeBatch();
    } catch (Exception e) {
        throw new QueryException("更改表错误: " + e.getMessage());
    }
}
```

以上的代码将用户操作过的对象转换成 SQL，例如，用户为表添加了若干个字段，那么就需要这样将 Field 对象转换成修改的 SQL。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\list\TableData.java

```
//返回全部的 ALTER TABLE ADD 字段语句,参数为用户添加的字段
private List<String> getAlterAddFieldSQL(List<Field> addFields) {
    List<String> result = new ArrayList<String>();
    for (Field f : addFields) {
        StringBuffer sql = new StringBuffer();
        sql.append("ALTER TABLE " + this.name).append(" ADD " + f.getFieldName())
            .append(" " + f.getType());
        if (!f.isAllowNull()) sql.append(" NOT NULL");
    }
}
```

```

        if (f.getDefaultValue() != null) sql.append(" DEFAULT " + f.getDefaultValue() + "");
        if (f.isAutoIncrement()) sql.append(" AUTO_INCREMENT");
        if (f.isPrimaryKey()) sql.append(", ADD PRIMARY KEY (" + f.getFieldName() + ")");
        result.add(sql.toString());
    }
    return result;
}

```

使用 **ALTER TABLE ADD** 来添加字段，那么如果修改了字段，就使用 **ALTER TABLE CHANGE**，删除字段使用 **ALTER TABLE DROP COLUMN**。如果添加外键，需要先为约束字段加索引（**ALTER TABLE ADD INDEX**），再使用 **ALTER TABLE ADD FOREIGN KEY**。如果是修改外键，就需要先将原来的外键删除，再重新建一次外键，重新建外键的方法与添加外键的顺序一致。删除外键使用 **ALTER TABLE DROP FOREIGN KEY**。

修改表无论从界面操作到拼装 SQL 都比较复杂，实现起来比较麻烦。在实现的过程中可以小步快跑，慢慢地一个一个来实现。

### 13.8.11 删除表

删除表实现十分简单，只需要执行 **DROP TABLE** 即可实现，在 **TableData** 中加入一个 **dropTable** 方法。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\list\TableData.java

```

//删除一个本类对应的表
public void dropTable() {
    StringBuffer sql = new StringBuffer();
    sql.append("DROP TABLE IF EXISTS " + this.name);
    Statement stmt = database.getStatement();
    stmt.execute(sql.toString());
}

```

表管理的大部分功能都已经实现，以上实现的这些功能中，大部分实现原理都十分简单，只是使用 **JDBC** 来执行一些 SQL 语句即可，但是界面的交互比较复杂，在实现的时候需要特别注意。本章中还有打开表、导出表数据的功能，将在下面的章节中描述。

## 13.9 数据浏览

在 13.6 中，我们实现了视图管理功能，在 13.8 中实现了表管理功能，在用户的实际操作中，当双击一个视图或者一个表的时候，我们需要将视图或者表对应的数据展现给用户，并在界面中显示，数据显示数据的界面已经在 13.2.3 中创建（**DataFrame** 类），如 13.2.3 中的图 13.4 所示。该界面包括刷新、降序和升序功能。

### 13.9.1 浏览数据

可以打开数据显示界面的地方有多个，包括打开视图、打开表、执行查询，视图在本章中使用一个 **ViewData** 对象来表示，表使用一个 **TableData** 对象表示，但是这些查询都是共一个界面，因此我们新建一个 **QueryObject** 的接口，让 **ViewData**（视图对象）和 **TableData** 去实现这个接口，该接口定义如下方法。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\QueryObject.java

```

public interface QueryObject {

```

```

//得到数据
ResultSet getDatas(String orderString);
//得到查询的名称
String getQueryName();
//返回查询的 SQL
String getQuerySQL(String orderString);
}

```

除了 **ViewData** 与 **TableData** 外，当执行一些 SQL 语句的时候，也需要将数据显示到 **DataFrame** 中，因此还需要建立一个 **QueryData** 的对象，表示一次查询所显示的数据对象，同样去实现 **QueryObject** 接口。**QueryObject** 接口只需要实现得到数据的方法，在 **ViewData** 和 **TableData**，只需要使用 **SELECT** 语句就可以从视图和表中得到相应的 **ResultSet** 对象。在本章中，**ViewData** 和 **TableData** 都保存一个 **Database** 对象，因此可以直接使用 **JDBC** 去执行，但是新建的 **QueryData** 对象就不属于任何的数据库，因此需要为该提供构造器。

代码清单：code\mysql-manager\src\org\crazyit\mysql\object\list\QueryData.java

```

//此次查询的 SQL 语句
private String sql;
//对应的数据库对象
private Database database;
public QueryData(Database database, String sql) {
    this.sql = sql;
    this.database = database;
}

```

编写完 **ViewData**、**TableData** 和 **QueryData** 对接口 **QueryObject** 的方法后，就可以将数据显示到 **DataFrame** 中。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\DataFrame.java

```

public DataFrame(QueryObject queryObject) {
    this.queryObject = queryObject;
    //省略其他代码
}

```

为 **DataFrame** 提供一个构造器，参数是接口 **QueryObject**，那么 **DataFrame** 就可以不用关心数据来源，只需要负责数据显示就可以了，这些数据来源只封装在一个 **QueryObject** 中，有可能是表的数据，也有可能是视图的数据。**DataFrame** 中处理数据的时候需要注意的是，**JTable** 的列也是动态的，它的列数由 **ResultSet** 来决定的，得到一个 **ResultSet** 的列数，可以使用 **ResultSetMetaData** 对象的 **getColumnCount** 方法来得到列数。

本章使用一个 **DataTable** 对象表示一个数据列表，该对象继承 **JTable**，在 13.2.3 中已经创建。新建一个 **DataColumn** 对象，表示一个数据列，新建一个 **DataCell** 对象，表示一个数据单元格对象。

**DataColumn** 包括如下属性。

```

//该列在 JTable 中的索引
private int index;
//该列的名称
private String text;

```

**DataCell** 包括如下属性：

```

//该单元格所在的行
private int row;
//该单元格所在的列
private DataColumn column;
//该单元格的值
private String value;

```

另外，需要为 **DataCell** 对象重写 **toString** 方法，返回该对象的 **value** 值。那么我们可以使用以下代



码来创建列表的数据。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\DataFrame.java

```
while (this.rs.next()) {
    DataCell[] data = new DataCell[columnCount];
    //遍历列，创建每一个单元格对象
    for (int j = 0; j < columnCount; j++) {
        //得到具体的某一列
        DataColumn column = this.columns.get(j);
        //创建单元格对象
        DataCell dc = new DataCell(i, column,
            this.rs.getString(column.getText()));
        data[j] = dc;
    }
    datas.add(data);
}
```

遍历一个 **ResultSet** 对象，遍历所有的列，最后构造一个 **DataCell** 对象，并存放到结果的集合中，注意结果集合是一个 **List** 对象，**List** 对象里面的元素是 **DataCell** 数组。最后在设置列表数据时候，可以 **DefaultTableModel** 的 **setDataVector** 方法将数据与列的信息设置到列表中。这里需要注意的是，不管以什么方式进入数据显示界面，都需要重新创建一个 **DataFrame** 对象。

### 13.9.2 刷新数据

在 **DataFrame** 中，已经保存有一个 **QueryObject** 的对象，如果需要对界面中的数据进行刷新，只需要重新读取一次数据，即执行 **QueryObject** 的 **getDatas** 方法，再重新将数据设置到列表中即可。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\DataFrame.java

```
//刷新数据
private void refresh() {
    //更新数据
    this.rs = this.queryObject.getDatas(this.orderString);
    //得到全部列
    this.columns = getColumns(this.rs);
    //设置数据到列表中
    this.model.setDataVector(getDatas(), this.columns.toArray());
    //设置每一列的宽
    setTableColumn(this.table);
}
```

除了刷新数据外，还有降序与升序的功能，当用户对数据进行了降序或者升序操作时，就可以调用 **refresh** 方法对列表进行刷新。

### 13.9.3 数据排序

我们在 13.2.3 中，已经实现了 **DataTable** 的整列选择（具体请看 13.2.3 章节），实现整列选择的时候，**DataTable** 中保存一个 **selectColumn** 的值，表示用户当前所选择的列索引，当用户选择了整列的时候，**selectColumn** 就等于具体的某列索引，因此就可以得到这一列的名称（数据库的字段名），然后再使用 **SQL** 中的 **ORDER BY** 语句对数据进行排序。

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\table\DataTable.java

```
//返回列的名称
private String getSelectColumnIdentifier() {
```

```

//得到选中列索引
int selectIndex = this.table.getSelectedColumn();
if (selectIndex == -1) return null;
DefaultTableColumnModel colModel = (DefaultTableColumnModel)this.table.getColumnModel();
return (String)colModel.getColumn(selectIndex).getIdentifier();
}
//降序
private void desc() {
    //得到字段名称
    String column = getSelectColumnNameIdentifier();
    //没有选中整列, 不排序
    if (column == null) return;
    this.orderString = column + " " + MySQLUtil.DESC;
    //刷新列表
    refresh();
}

```

以上代码实现降序。在接口 `QueryObject` 的 `getDatas` 方法中, 需要提供参数 `orderString`, 即排序语句, 因此只需要重新执行 `QueryObject` 的 `getDatas` 方法即可得到排序后的数据。升序的实现与降序一致, 只是使用 SQL 的 `ORDER BY ASC` 即可实现。

## 13.10 执行SQL

执行 SQL 的界面已经在 13.2.8 中实现, 只是一个简单的文本域让用户输入 SQL, 然后提供一个执行 SQL 与保存 SQL 的功能, 由于我们已经实现了数据浏览的显示, 因此实现起来十分简单。执行 SQL 的界面在本章中为 `QueryFrame` 类。

### 13.10.1 运行SQL

在 13.2.8 中, 用户只需要输入相关的 SQL 语句, 就可以执行该 SQL 语句, 如 13.2.8 中的图 13.10 所示。如果用户输入的是 `INSERT`、`UPDATE` 等语句, 那么就可以将执行结果直接使用普通的提示框显示出来, 如果用户输入的是 `SELECT` (查询语句)、`CALL` (调用存储过程或者函数) 语句, 那么就需要将查询封装成一个 `QueryData` 对象 (13.9.1 中创建), 再将 `QueryData` 对象作为构造参数传递给 `DataFrame`, `QueryData` 是 `QueryObject` 接口的其中一个实现。

代码清单: `code\mysql-manager\src\org\crazyit\mysql\ui\QueryFrame.java`

```

//查询
private void query() {
    //得到 SQL
    String sql = this.editArea.getText();
    try {
        //封装一个 QueryData 对象
        QueryData queryData = new QueryData(this.database, sql);
        //显示 DataFrame
        DataFrame dataFrame = new DataFrame(queryData);
        dataFrame.setVisible(true);
    } catch (Exception e) {
        e.printStackTrace();
        showMessage(e.getMessage(), "错误");
    }
}

```

```
    }
}
```

以上的 query 方法，只有当用户输入有 **SELECT** 或者 **CALL** 关键字的时候才调用，其他情况则直接弹出执行结果的提示（成功与否）。

### 13.10.2 保存SQL

保存 SQL，只是将用户输入的 SQL 语句保存到一份文件中即可。从界面得到用户输入的数据，然后提供一个文件选择器让用户选择，最后使用 IO 流将内容输入到文件就可以实现。

代码清单：code\mysql-manager\src\org\crazyit\mysql\util\FileUtil.java

```
//创建文件并将 content 写到 File 中
public static void writeToFile(File file, String content) {
    //创建新文件
    createNewFile(file);
    //写入文件
    FileWriter writer = new FileWriter(file);
    writer.write(content);
    writer.close();
}
```

代码清单：code\mysql-manager\src\org\crazyit\mysql\ui\QueryFrame.java

```
//写入文件
public void writeToFile(File file) {
    String content = this.editArea.getText();
    //将内容写到文件中
    FileUtil.writeToFile(file, content);
}
```

执行 SQL 的功能已经完成，这是用户输入 SQL 再运行的一种操作形式，用户还有另外一种操作形式，就是通过导入文件来执行一份 SQL 文件，下一小节将讲解如何实现 SQL 文件的导入与导出。

## 13.11 SQL文件的导入与导出

SQL 文件的导入与导出，包括数据库的导入与导出、表的导出。当用户选择了某个数据库的时候，就提供鼠标右键菜单，让用户可以执行数据库的导出与导入操作。当用户选择了某个表的时候，就可以提供鼠标右键菜单，提供导出功能。右键菜单已经在 13.2.9 与 13.2.10 中实现。

### 13.11.1 执行SQL文件

进入管理器的第一个界面，就需要用户选择 MySQL 的安装目录，根据用户所选择的目录，我们就可以找到 MySQL 安装目录下面的 bin 目录，然后找到 mysqldump 与 mysql 这两个命令。mysqldump 命令主要用于导出 SQL 文件，mysql 命令主要用于执行 SQL 文件。

在 13.3 小节中，我们实现了用户选择 MySQL 安装目录的功能，并且将用户选择的目录存放到一个 GlobalContext 的对象中，那么如果需要使用 mysql 命令执行 SQL 文件，直接拼装命令执行即可。至于执行的方式，将在下面讲解。

例如 MySQL 中存在一个数据库，如果需要执行某份 SQL 文件，就需要执行以下语句：

```
"MySQL 安装目录\bin\mysql" -u 用户名 -p 密码 -h 连接 IP -D 数据库名称 < "SQL 文件绝对路径"
```

使用 mysql 命令执行 SQL 文件，mysql 命令有许多的参数，以上语使用了 -u、-p、-h 和 -D 参数，

-u 参数表示数据库的用户名，-p 表示密码，-h 表示连接的服务器 IP，-D 表示需要执行文件的数据库，拼装好以上的语句后，可以使用 Runtime 类的 exec 方法执行。

注意：这里需要特别说明的是，如果 MySQL 安装目录有空格，或者 SQL 文件的绝对路径有空格，首先需要将 mysql 命令（包括安装目录）使用双引号括起来，再将 SQL 文件的绝对路径使用双引号括起来，但是直接使用 Runtime 的 exec 执行仍然会出错，我们可以将拼装的那句命令，使用 IO 流写入到一份 .bat 文件中，然后使用 Runtime 的 exec 方法执行：“cmd /c bat 文件所在”可消除错误。当然，这是在 Windows 操作系统下，如果使用 Linux 的话，可以生成 sh 文件。

### 13.11.2 导出数据库与表

导出数据库与执行 SQL 文件一样，使用 mysqldump 命令即可实现。mysqldump 语句格式如下：

```
"MySQL 安装目录\bin\mysqldump" -u 用户名 -p 密码 -h 连接 IP --force --databases 数据库名 > "导出的 SQL 保存目录"
```

以上使用了 mysqldump 集合的 -u、-p、-h、--force 和 --databases 参数，-u、-p 和 -h 分别代表用户名、密码和数据库服务器 IP，--force 参数表示发生错误将继续执行，--database 参数表示需要导出的数据库名称。导出 SQL 文件与执行 SQL 文件一样，都是将拼装的命令使用 IO 流写到一份 bat 或者 sh 文件中，再使用 Runtime 的 exec 方法执行。

导出表与导出数据库实现一致，只需要在导出数据库的参数的基础上，再加上 --tables 参数来声明需要导出的表即可。多个表之间使用空格将表名分隔。导出表使用的 mysqldump 语句格式如下：

```
"MySQL 安装目录\bin\mysqldump" -u 用户名 -p 密码 -h 连接 IP --databases 数据库名称 --tables 表一 表 N > "导出的 SQL 文件保存目录"
```

在本章中，处理 SQL 文件的导入与导出由 BackupHandler 接口完成，该接口有一个 BackupHandlerImpl 的实现类，已经对 SQL 文件的导出和导出进行实现，这些实现只是拼装一些语句，真正执行这些语句由 CommandUtil 中的 executeCommand 方法执行，该方法提供了 Windows 下的实现（生成一份 bat 文件并执行、最后删除）。

## 13.12 本章小节

本章实现了一个 MySQL 管理器，这个管理器中有多个功能，包括数据库元素的管理、数据浏览与 SQL 文件的导出和导出。我们可以在管理器实现的过程中，了解 MySQL 管理器的实现原理。实现 MySQL 管理器功能并不困难，困难的是一些界面的交互，特别是表管理界面。本章的这个 MySQL 管理器与一些流行的管理器有着部分的区别，例如我们在进入管理器的时候，需要用户选择 MySQL 的安装目录，目的是为了使用 MySQL 的一些内置命令，但是例如平常使用的 Navicat，它并不需要知道 MySQL 的安装目录（或者根本不需要安装 MySQL），使用程序来导出 SQL 文件。当用户需要导出 SQL 文件的时候，我们也可以使用程序来对数据库进行分析，再写到 SQL 文件中，这样也是一种实现途径。本章的目的并不是与这些商业管理器竞争，而是让读者了解管理器实现的原理。希望大家能在本章的基础上，开发出功能更加强大的 MySQL 管理器。

## 第 14 章 自己开发 IoC 容器

### 14.1 IoC 简介

在平时的开发中，当我们正在编写的某一个类需要用到另外的一个类（组件）的时候，我们都需要通过 `new` 的关键来创建该类的实例，那么有没有更好的方式，可以不在我们的代码中直接去 `new` 这个类，就可以得到该实例呢？IoC 的概念很好的帮助我们解决了这个问题，当我们需要在一个类中使用另外的类时，可以通过一些配置来得到该类的实现。IoC 是 **Inversion of Control** 简称，又称控制反转，一个类需要另外一个类的实例，可以通过某个容器获得，而不在类的内部创建，得到什么样的实例，由容器去决定，而不是由该类决定。

搞清楚了控制反转的概念后，我们再来了解什么叫依赖注入。依赖注入（**Dependency Injection**）与控制反转是同一个概念，我们创建某个类的实例由 IoC 容器完成，如果该类需要使用另外的类实例，那么可以在 IoC 容器中向该类注入需要被调用的实例。

无论是控制反转还是依赖注入，采用这种方式来创建类的实例，可以使得我们的代码更加清晰明了，将各个类之间的依赖关系反映到 IoC 容器中，可以更动态、灵活和透明的管理各个对象。在 J2EE 领域中使用 IoC 的概念，可以将各个功能不同的组件统一放到 IoC 容器中，程序员只需要关注各个组件的实现，而不需要关注各个组件的依赖的关系。常用的 IoC 容器有 Spring 的 IoC 容器、webwork 的 IoC 容器、google-guice、apache 的 HiveMind 等。

笔者面试过一些工作了几年程序员，这些程序员很多都说自己精通 Spring，精通 IoC 容器，但是如果一问 IoC 容器是如何帮我们创建类实例的，很多都不知道如何回答。在下面章节中，我们将通过编写一个简单的 IoC 容器，来说明 IoC 容器是如何帮我们创建类的实例、如何实现依赖注入，最后并通过一个整合的例子，来说明 IoC 容器给我们带来什么样的好处。

### 14.2 使用技术简介

在本章开发 IoC 容器时，所涉及的技术有如下几种：

- ☐ Java 的反射机制
- ☐ dom4j
- ☐ Junit

#### 14.2.1 Java 反射简介

在 Java 运行时环境中，如果我们需要得到某一个类的具体信息，那么就可以使用 Java 的反射机制，该机制可以让我们动态的得到某个类型的属性、构造器和方法。Java 的反射机制可以在运行时构造某一个类的实例，在运行时调用任意一个对象的方法。

在本章中，我们需要在运行时去动态的加载配置文件，再根据这些配置文件去创建某一个类的实现，并为这些实例设置相应的属性，因此我们需要使用 Java 的反射机制。

### 14.2.2 dom4j

dom4j 是一个常用的 XML 解析项目，该项目是一个易用的开源项目，它应用于 Java 平台，支持使用 Dom、SAX 来解析 XML 文件。Dom4j 的使用非常简单，只需要使用该项目所提供的解析类，就可以轻松的读取到相应的 XML 文件，并可以得到这些 xml 的相关内容。本章中我们采用了 XML 文件作为我们的 IoC 容器的配置文件，因此使用 dom4j 就非常的合适。

### 14.2.3 Junit

Junit 是一个单元测试框架，供 Java 程序员编写单元测试。在本章中，我们每编写一个功能点，就需要对该功能进行单元测试，一来可以展示我们所编写程序的效果，二来可以保证我们的程序的质量。如果站在 XP（极限编程）的角度来讲，编写测试可以让我们的代码更加健壮，更不惧任何的变更，对我们项目的发展有长远的利益。本章的重点是 IoC 容器，使用这个测试框架，目的是展示我们代码的效果，我们当前所使用的是 Junit4。

## 14.3 确定配置文件内容、编写DTD

在编写 IoC 容器之前，我们需要编写 XML 文件，这些 XML 文件用来定义人们 IoC 容器的一些配置，例如声明我们需要创建哪些对象（bean），为这些对象（bean）提供名字和类名，让我们的 IoC 容器根据这些信息去创建相应的 bean。除了定义 bean 的名字和类名外，还需要定义一些其他的属性，例如该 bean 是否为单态，是否需要延迟加载等。确定了配置文件的内容后，我们开始着手编写 DTD 文件。

### 14.3.1 声明bean

在整个 IoC 容器中，每一个 bean 代表具体的某个类，因此我们的根节点为 beans，beans 下有多个 bean，可以让我们配置具体的某个类。XML 文件的具体配置如下：

```
<bean>
  <bean></bean>
  ...
</beans>
```

定义好了 bean 节点好，我们需要为 bean 节点指定一个名字和类名，表示这个 bean 所对应的名称与类型，这些我们的 IoC 容器得到这些配置后，就可以帮我们创建这些类的实例。具体的配置如下：

```
<beans>
  <bean id="myDate" class="java.util.Date"></bean>
</beans>
```

以上的 bean 配置表示我们在 IoC 容器中创建了一个 myDate 的 bean，该 bean 的类型是 java.util.Date。

### 14.3.2 声明单态的bean

定义一个 bean 除了需要显式声明该 bean 的名称和类型外，还需要告诉我们的 IoC 容器，该 bean 是否为单态，如果该 bean 是单态的话，那么 IoC 容器在启动的时候，就只会为该 bean 创建一个实例，并将该实例缓存起来，如果该 bean 配置成非单态的话，那么就不进行缓存，每次请求这个 bean 的时候，就重新创建一个。我们为 bean 的配置添加一个 singleton 属性，用来声明 bean 是否为单态。

```
<beans>
  <bean id="myDate" class="java.util.Date" singleton="false"></bean>
</beans>
```

以上配置的黑体部分，我们配置了 **myDate** 这个 **bean** 为非单态，在一般的情况下，我们可以不指定 **singleton** 属性，让 **singleton** 属性在不显式声明的情况下的值为 **true**，给配置节点提供默认值，在 14.4 准备 DTD 文件中有详细说明。

### 14.3.3 声明延迟加载

当 IoC 容器启动的时候，容器是否需要马上创建这个 **bean**，我们可以为配置文件提供一个属性，让容器知道我们在容器初始化的时候，是否需要创建。我们为 **bean** 提供一个 **lazy-init** 的属性：

```
<bean id="myDate" class="java.util.Date" lazy-init="true"></bean>
```

以上的配置代码中，声明了 **myDate** 需要延迟加载，为 **bean** 节点加了这个属性后，**beans** 节点也可以提供一个 **default-lazy-init** 的属性，表示这份配置文件下的所有 **bean**，如果 **lazy-init** 属性的值为 **default** 的时候，就使用 **beans** 节点（根节点）的 **default-lazy-init** 所配置的值：

```
<beans default-lazy-init="true">
  <bean id="myDate" class="java.util.Date" lazy-init="default"/>
</beans>
```

以上的配置，如果 **myDate** 的 **bean** 声明为 **default** 的话，那么就按照 **beans** 的 **default-lazy-init** 属性来决定是否需要延迟加载。如果对 **bean** 的 **lazy-init** 属性不指定值的话，那么就为这个 **lazy-init** 属性指定为 **default**，表示这个 **bean** 是否延迟加载取决于 **beans** 的配置，而 **beans** 的 **default-lazy-init** 属性不显式指定的话，可以使用 **false** 作为默认值。

### 14.3.4 声明设值注入到bean的属性

IoC 容器除了可以帮我们创建 **bean** 之外，还可以帮助我们将对应的属性设置到该 **bean** 的实例里面，我们需要为每个 **bean** 注入一些属性，让该 **bean** 的实例得到这些属性，这些属性有可能是一些普通的值，也有可能是另外的定义的 **bean**。在这里我们使用设值注入，设值注入，就是创建了 **bean** 的实例后，再获得该 **bean** 配置的一些属性，通过该 **bean** 里面的 **setter** 方法，设值到该 **bean** 的实例中。

```
<beans>
  <!--定义一个 student 的 bean -->
  <bean id="student" class="org.crazyit.Student">
    <!--为 Student 类中的 school 属性注入 school 的 bean -->
    <property name="school">
      <ref bean="school"/>
    </property>
  </bean>
  <bean id="school" class="org.crazyit.School"/>
</beans>
```

以上的配置中的黑体部分，我们为 **student** 的 **bean** 注入了一个 **school** 的 **bean**，这样，**school** 和 **student** 这两个 **bean** 就产生了依赖关系，这样的注入我们就叫依赖注入，以上的注入方式是依赖注入中的设值注入。那么我们的配置文件中需要为 **bean** 加入一个 **property** 的子节点，该节点里面有一个 **name** 属性，**property** 节点下面有一个 **ref** 子节点，**ref** 子节点指向某一个容器中的 **bean**。除了注入另外定义的 **bean** 外，还可以向 **bean** 中注入一些普通的属性：

```
<bean id="student" class="org.crazyit.Student">
  <!--为 Student 类中的 school 属性注入 school 的 bean -->
  <property name="age">
    <value type="java.lang.Integer">18</value>
  </property>
</bean>
```

```

    </property>
</bean>

```

以上的配置表示向 `student` 这个 bean 中注入了一个 `Integer` 的值, 对应 `Student` 的 `age` 属性。因此, `property` 节点下面可以出现 `ref` 节点和 `value` 节点, 但是, 每一个 `property` 只能出现一次 `ref` 节点或者 `value` 节点, 因为我们一个 `property` 只会设置一个属性。

### 14.3.5 声明构造注入到bean的属性

14.3.4 小节中, 我们定义配置文件中为 bean 提供 `property` 节点进行设值注入, 本小节我们定义构造注入的节点。构造注入, 也就是通过 bean 中定义的构造参数, 在创建这个 bean 的实例时, 就将这些参数通过调用该 bean 的构造器, 将这些配置的属性传递给该 bean。

```

<bean id="student" class="org.crazyit.Student">
    <constructor-arg>
        <ref bean="school">
    </constructor-arg>
</bean>

```

以上配置的黑体部分, 声明了创建 `student` 这个 bean 的时候, 我们需要为 `Student` 类提供一个构造器, 构造器的参数为 `School` 类型, 那么 IoC 容器在创建时, 就会根据 `constructor-arg` 声明的属性, 去调用 `Student` 的构造器。与 14.3.4 中的设值注入一样, 除了可以提供 `ref` 元素外, 还可以提供 `value` 元素, 构造注入一些另外的普通属性。

```

<bean id="student" class="org.crazyit.Student">
    <constructor-arg>
        <value type="java.lang.Integer">20</value>
    </constructor-arg>
</bean>

```

这样的配置, 我们就可以为 `Student` 进行构造注入, 并在创建 bean 的实例时, 提供一个 `Integer` 类型的构造参数。

### 14.3.6 自动装配

自动装配, 就是不需要指定 bean 的属性, 从 IoC 容器中查找 bean 所需要的属性, 将这些查找到的 bean 以设值注入的方式依赖注入到目标的 bean 中。

```

<bean id="student" class="org.crazyit.Student" autowire="byName"></bean>
<bean id="school" class="org.carayit.School"></bean>

```

以上的 `student` 提供了 `autowire` 属性, 该属性值为 `byName`, 表示根据 bean 的名称自动注入到 `student` 中, 如果 `Student` 类中有一个 `setSchool` 的 setter 方法, 并且参数为 `School` 对象, 那么容器就需要自动的将 `School` 的 bean 通过设值注入设置到 `Student` 中。与延迟加载一样, 也可以为 `beans` 节点提供一个 `default-autowire` 属性, 声明该 `beans` 根节点下面所有的 bean 节点, 如果 `autowire` 的值为 `default`, 那么就可以进行自动装配。

```

<beans autowire="byName">
    <bean id="student" class="org.crazyit.Student" autowire="default"></bean>
    <bean id="school" class="org.carayit.School"></bean>
</beans>

```

在本例中 `autowire` 的值我们定义为只允许为 `no` 或者 `byName`。如果 `autowire` 的值为 `no` 的话, 表示不需要自动装配。bean 的 `autowire` 属性默认值为 `default`。

确定了 IoC 的配置文件需要配置的内容, 那么接下来, 我们就可以根据上述的配置内容, 编写相关的约束文件 (DTD)。



### 14.3.7 准备 DTD 文件

文档定义类型 (DTD) 可以定义合法的 XML 文档构建模块, 对我们所编写的一些 XML 文件进行约束, 也就是说, 当一份 XML 指定了某份 DTD 文件时, 那么该 XML 文件就必须遵守该 DTD 文件的约束。简单的说, DTD 就是一种对 XML 文件定制规范。

当我们需要为一份 XML 定义某份 DTD 规范的时候, 在 XML 文件头中声明:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//CRAZYIT//DTD BEAN//EN"
"http://www.crazyit.org/beans.dtd">
```

以上的 XML 文件头, 声明了该份 XML 由 <http://www.crazyit.org/beans.dtd> 这一份 dtd 来约束, 每个元素都需要遵守该份 DTD 中所声明的文档结构。

根据 14.3.1 到 14.3.6 里面定义的配置文件的内容, 我们可以开始编写 DTD 文件。以下为我们的 IoC 容器配置文件的 DTD 文件:

```
<!ELEMENT beans (
    bean*
)>
<!ATTLIST beans default-lazy-init (true | false) "false">
<!ATTLIST beans default-autowire (no | byName) "no">
<!-- 指定 bean 元素的子元素 -->
<!ELEMENT bean (
    (constructor-arg | property)*
)>
<!-- 指定 bean 元素的属性值 -->
<!ATTLIST bean id CDATA #REQUIRED>
<!ATTLIST bean class CDATA #REQUIRED>
<!ATTLIST bean lazy-init (true | false | default) "default">
<!ATTLIST bean singleton (true | false) "true">
<!ATTLIST bean autowire (no | byName | default) "default">
<!-- 声明 constructor-arg 子元素 -->
<!ELEMENT constructor-arg (
    (ref | value | null)
)>
<!-- 声明 property 元素的子元素 -->
<!ELEMENT property (
    (ref | value | null)?
)>
<!-- 声明 property 的属性 -->
<!ATTLIST property name CDATA #REQUIRED>
<!-- 声明 property 的属性 -->
<!ATTLIST value type CDATA #REQUIRED>
<!-- 声明 ref 元素 -->
<!ELEMENT ref EMPTY>
<!-- 声明 ref 的属性 -->
<!ATTLIST ref bean CDATA #REQUIRED>
<!-- 声明 value 元素 -->
<!ELEMENT value (#PCDATA)>
```

通过以上的代码, 那么该份 DTD 的各个节点可以概括为如下几个:

- ❑ **beans 节点:** 根节点为 beans, beans 的属性有 default-lazy-init 和 autowire, 这两个属性的默认值 (不需要显式提供) 为 false 和 no。

- ❑ **bean 节点**: beans 下面有多个 bean 节点, bean 节点必须要显式提供 id 和 class 属性, 可以不必显式提供 lazy-init, singleton 和 autowire 属性, lazy-init 的默认值为 default, 表示该值由 beans 的 default-lazy-init 来决定, singleton 的默认值为 true, 而且只允许有 true 和 false 两个值, autowire 属性的默认值是 default, 由 beans 的 default-autowire 来决定。
  - ❑ **construct-arg 节点**: ref、value 和 null 都可以作为该节点的子节点, 该节点没有属性。
  - ❑ **property 节点**: ref、value 和 null 都可以作为该节点的子节点, property 节点有一个 name 属性, 而且是必须指定的。
  - ❑ **ref 节点**: 该节点没有子节点, 只有一个必须指定的 bean 属性。
  - ❑ **value 节点**: value 节点只有一个 type 属性, 用于指定该值的类型。
- 编写完 DTD 文件后, 那么我们可以在 XML 的文件头中声明 DTD。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//CRAZYIT//DTD BEAN//EN"
    "我们编写的 DTD 文件的绝对路径">
```

以上的 XML 文件头, 表示我们使用某份 DTD 文件作为 XML 的约束, 指定 DTD 文件路径时, 我们暂时使用绝对路径, 指向我们上面编写的那份 DTD 文件, 在下面的章节中, 将会讲解如何将绝对路径变成 url。

## 14.4 读取XML文件

在 14.2 中, 我们已经确定了使用 dom4j 来读取 XML 文件, 我们需要明白, 当在创建一个 IoC 容器的时候, 我们就开始读取 XML 文件, 当取完之后, 可以将 XML 中我们所需要的信息缓存起来, 也就是进行一次读取即可, 如果进行多次读取, 那么将会影响性能。

在使用 dom4j 之前, 我们需要下载 dom4j 的包, 下载完包之后, 可以将包放到项目的环境变量中。dom4j 的包可以从 sourceforge 中下到, 下载的地址为: <http://sourceforge.net/projects/dom4j/files/>, 本例所使用的 dom4j 版本是 1.6.1。

### 14.4.1 加载XML文件

在本小节开头, 我们已经确定了只读取一次 XML, 那么我们就新建一个类, 用于读取 XML, 并将读取到的 Document 对象缓存。建立 DocumentHolder 接口。

代码清单: code\IoC\main\org\crazyit\ioc\xml\DocumentHolder.java

```
public interface DocumentHolder {
    //根据文件的路径返回文档对象
    Document getDocument(String filePath);
}
```

DocumentHolder 接口只有一个方法, 用于获得一个 Document 对象, 参数是 XML 文件的路径。为该接口新建一个实现类 XmlDocumentHolder。

代码清单: code\IoC\main\org\crazyit\ioc\xml\XmlDocumentHolder.java

```
public class XmlDocumentHolder implements DocumentHolder {
    //新建一个 Map 对象, 用于保存读取到的多份 XML 文件
    private Map<String, Document> docs = new HashMap<String, Document>();
    public Document getDocument(String filePath) {
        Document doc = this.docs.get(filePath);
        if (doc == null) {
```

```

        this.docs.put(filePath, readDocument(filePath));
    }
    return this.docs.get(filePath);
}
//根据文件路径读取 Document
private Document readDocument(String filePath) {
    try {
        //使用 SAXReader 来读取 xml 文件
        SAXReader reader = new SAXReader(true);
        //使用自己的 EntityResolver
        reader.setEntityResolver(new IoEntityResolver());
        File xmlFile = new File(filePath);
        //读取文件并返回 Document 对象
        Document doc = reader.read(xmlFile);
        return doc;
    } catch (Exception e) {
        e.printStackTrace();
        throw new DocumentException(e.getMessage());
    }
}
}

```

以上代码中，我们使用了一个 Map 对象来保存多个 Document 对象，当外界使用 `getDocument` 方法的时候，直接从 Map 对象中获取该 Document 对象，如果取到就返回，如果取不到该对象，就通过私有方法 `readDocument` 来读取。注意以上代码的黑体部分，我们使用了自定义的 `EntityResolver` 对象，`IoEntityResolver` 的代码如下。

代码清单：code\IoC\main\org\crazyit\ioc\xml\IoEntityResolver.java

```

public class IoEntityResolver implements EntityResolver {
    public InputSource resolveEntity(String publicId, String systemId)
        throws SAXException, IOException {
        //先从本地寻找 dtd
        if ("http://www.crazyit.org/beans.dtd".equals(systemId)) {
            InputStream stream = IoEntityResolver.class.
                getResourceAsStream("/org/crazyit/ioc/beans/beans.dtd");
            return new InputSource(stream);
        } else {
            return null;
        }
    }
}

```

在 `IoEntityResolver` 中，如果 `systemId` 是一个我们定义的 url，那么就先从本地读取该 DTD 文件，如果找不到该 DTD 文件，则自动到网络上去寻找。那么我们可以在 XML 的文件头中声明这样的地址：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//CRAZYIT//DTD BEAN//EN"
    "http://www.crazyit.org/beans.dtd">

```

以上 XML 文件头的信息中的黑体部分，就是 `IoEntityResolver` 类中的 `resolveEntity` 方法中的 `systemId` 参数。表面上好像 XML 文件从网络去寻找对应的 DTD 文件，但事实上是从本地的编译路径中寻找。

编写代码后，我们可以编写一个单元测试查看我们的代码是否准确。如果你有良好的编程习惯，那么最好还是先编写你的单元测试，再去实现你的代码。我们可以从以下地址得到 Junit，本例所使用的版

本是 Junit4: <http://sourceforge.net/projects/junit/files/junit/>。新建一个测试类 `XmlHolderTest`，该类继承于 Junit 的 `TestCase`。

代码清单: `code\IoC\test\org\crazyit\ioc\xml\XmlHolderTest.java`

```
public class XmlHolderTest extends TestCase {
    //需要测试的对象
    XmlDocumentHolder holder;
    protected void setUp() throws Exception {
        holder = new XmlDocumentHolder();
    }
    protected void tearDown() throws Exception {
        holder = null;
    }
    public void testGetDoc() {
        String filePath = "test/resources/XmlHolder.xml";
        //获得 Document 对象
        Document doc = holder.getDocument(filePath);
        Element root = doc.getRootElement();
        System.out.println(root.getName());
        //重新获取一次, 判断两个 Document 对象是否一致
        Document doc2 = holder.getDocument(filePath);
        System.out.println(doc);
        System.out.println(doc2);
    }
}
```

在测试类中, 由于该类继承了 `TestCase` 类, 因此可以重写 `setUp` 和 `tearDown` 方法, 表示在测试前和测试后会执行的方法, 以上代码中, 我们测试前创建了 `XmlDocumentHolder` 对象, 测试后将其销毁。如果要运行某个测试方法, 该方法需要用 `test` 作为方法名前缀, 在 Junit4 中, 提供了 `@Test` 的注解来运行单元测试, 可以不必使用 `test` 作为方法名的开始。在本例中, 我们可以不使用 Junit 作为测试工具, 可以使用 `main` 方法进行测试, 并查看结果。

编写了测试类之后, 我们需要编写一份 XML 文件给测试类去读取, 以下是测试所使用的 XML 文件内容, `XmlHolder.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//CRAZYIT/DTD BEAN//EN" "http://www.crazyit.org/beans.dtd">
<beans>
    <bean id="test" class=" org.crazyit.ioc.xml.TestObject1"></bean>
</beans>
```

提供了 XML 文件后, 可以直接运行测试类, 由于本章的重点不是 Junit, 因此如何运行 Junit 的测试, 可以查看 Junit 的相关文档。运行 `XmlHolderTest` 的结果如下:

```
beans
org.dom4j.tree.DefaultDocument@7ffe01 (其他省略)
org.dom4j.tree.DefaultDocument@7ffe01 (其他省略)
```

我们可以看到打印出来的结果与我们所想要的结果一样, 其实使用 Junit 可以直接写断言, 在这里省略。这样, 读取 XML 文件的程序已经实现了, 但是, 我们的 IoC 容器并不是只需要一个 `Document` 对象就够的, 我们还需要得到每一个 `bean` 所对应的 `Element` 对象。接下来编写 `Element` 对象的读取。

## 14.4.2 读取Element

在 14.4.1 中, 我们得到 XML 文件对应的 `Document` 对象, 那么我们可以根据这些 `Document` 对象得到所有的 `Element` 对象, 也就是配置文件中的每一个 `bean`。

新建接口 `ElementLoader`。

代码清单：code\IoC\main\org\crazyit\ioc\xml\ElementLoader.java

```
public interface ElementLoader {
    //加入一份 doc 的所有 Element
    void addElements(Document doc);
    //根据元素 id 获得 Element 对象
    Element getElement(String id);
    //返回全部的 Element
    Collection<Element> getElements();
}
```

我们提供一个 `ElementLoader` 的接口，用来获得 `Document` 的所有 `Element`，我们在 14.3 中已经定义了 DTD 规范，因此，不用去担心 XML 不符合我们的要求。

以上为 `ElementLoader` 的实现 `ElementLoaderImpl`。

代码清单：code\IoC\main\org\crazyit\ioc\xml\ElementLoaderImpl.java

```
public class ElementLoaderImpl implements ElementLoader {
    //提供一个 Map 来保存, Map 的 key 为 bean 元素的 id, value 为 bean 对应的 Element 对象
    private Map<String, Element> elements = new HashMap<String, Element>();
    public void addElements(Document doc) {
        //读取根节点 beans, 再得到所有的 bean 节点
        List<Element> eles = doc.getRootElement().elements();
        for (Element e : eles) {
            //得到 bean 的 id 属性
            String id = e.attributeValue("id");
            //添加到 map 中
            elements.put(id, e);
        }
    }
    public Element getElement(String id) {
        return elements.get(id);
    }
    public Collection<Element> getElements() {
        return this.elements.values();
    }
}
```

与 `XmlDocumentHolder` 一样，`ElementHoderImpl` 也提供一个 `Map` 来保存所有的 `Element`，当调用 `addElement` 方法的时候，就将参数中的 `Document` 里面所有的 `Element` 元素（bean 节点）都放到 `Map` 中，这样无形中也对我们的 XML 文件进行了另外一种约束，在整个 IoC 容器中，bean 的 id 必须是唯一的，否则的话，先读取的 `Element` 对象，会被后读取的名字相同的 `Element` 覆盖。

提供测试用的 XML 文件：

```
<beans>
    <bean id="test1" class=" org.crazyit.ioc.xml.TestObject1" lazy-init="true"></bean>
    <bean id="test2" class=" org.crazyit.ioc.xml.TestObject1" lazy-init="true"></bean>
    <bean id="test3" class=" org.crazyit.ioc.xml.TestObject1"></bean>
    <bean id="test4" class=" org.crazyit.ioc.xml.TestObject1"></bean>
</beans>
```

编写测试代码：

```
public void testGetElements() {
    String filePath = "test/resources/ElementLoader.xml";
    Document doc = holder.getDocument(filePath);
}
```

```
elementLoader.addElements(doc);  
//得到 bean id 为 test1 的 Element 对象  
Element e = elementLoader.getElement("test1");  
System.out.println(e);  
}
```

运行可以看到打印结果如下：

```
org.dom4j.tree.DefaultElement@117a8bd （省略其他）
```

以上结果表示我们已经将 **Element** 对象放到 **Map** 中。再调用 **getElements** 方法，打印集合的大小，可以看到结果为 4。

### 14.4.3 解析Element

我们得到 **Element** 对象后，还需要对 **Element** 进行解析，得到相关的信息。例如，我们需要判断一个 **Element** 所对应的 **bean** 是否为单态的，是否需要延迟加载等信息，都是需要通过读取 **Element** 来得到这些信息的。新建 **ElementReader** 接口。

代码清单：code\IoC\main\org\crazyit\ioc\xml\ElementReader.java

```
public interface ElementReader {  
    //判断一个 bean 元素是否需要延迟加载  
    boolean isLazy(Element element);  
    //获得一个 bean 元素下的 constructor-arg  
    List<Element> getConstructorElements(Element element);  
    //得到元素属性为 name 的属性值  
    String getAttribute(Element element, String name);  
    //判断一个元素是否为单态  
    boolean isSingleton(Element element);  
    //获得一个 bean 元素下所有 property 元素  
    List<Element> getPropertyElements(Element element);  
    //返回一个 bean 元素对应的 Autowire 对象  
    Autowire getAutowire(Element element);  
    //获取 bean 元素下所有 constructor-arg 的值(包括 value 和 ref)  
    List<DataElement> getConstructorValue(Element element);  
    //获取 bean 元素下所有 property 元素的值(包括 value 和 ref)  
    List<PropertyElement> getPropertyValue(Element element);  
}
```

**ElementReader** 接口包括读多的方法，都是通过一个 **Element** 获得该 **Element** 对应的 **bean** 的相关信息：

- ❑ **isLazy**：判断一个 **Element** 的配置是否需要延迟加载。
- ❑ **getConstructorElements**：获得一个 **Element** 下所有的 **constructor-arg** 节点，并返回该节点对应的 **Element** 集合。
- ❑ **getAttribute**：获得一个 **Element** 的属性，第二个参数 **name** 为属性名字。
- ❑ **isSingleton**：判断一个 **Element** 对应的 **bean** 是否为单态。
- ❑ **getPropertyElements**：获得一个 **bean** 节点下面所有的 **property** 节点，并以 **Element** 集合返回。
- ❑ **getAutowire**：判断一个 **Element** 对应的 **bean** 是否需要自动装配，返回一个 **Autowire** 对象，该对象在下面将会描述。
- ❑ **getConstructorValue**：获得一个 **bean** 下所有的 **constructor-arg** 的值集合，这些值包括 **ref** 的值和 **value** 的值。
- ❑ **getPropertyValue**：获得一个 **bean** 下所有的 **property** 的值集合，这些值包括 **ref** 的值和 **value**

的值。

**Autowired** 接口。代码清单：code\IoC\main\org\crazyit\ioc\xml\autowire\Autowired.java

```
public interface Autowire {  
    //返回自动装配的配置值  
    String getValue();  
}
```

**Autowired** 代表一个 bean 的 **autowire** 属性，我们将该属性封装成一个对象，那么如果 **autowire** 的值为 **byName**，那么我们再提供一个 **Autowired** 的实现类 **ByNameAutowired**。

代码清单：code\IoC\main\org\crazyit\ioc\xml\autowire\ByNameAutowired.java

```
public class ByNameAutowired implements Autowire {  
    private String value;  
    public ByNameAutowired(String value) {  
        this.value = value;  
    }  
    public String getValue() {  
        return value;  
    }  
}
```

这个对象是接口 **Autowired** 的一个实现，如果返回这个对象的话，表明 **autowire** 的属性是 **byName**，同样的，提供一个 **NoAutowired** 的类，实现与 **ByNameAutowired** 一样，这两个对象只是用于判断值的类型。

注意接口中 **getConstructorValue** 方法，返回一个 **DataElement** 对象，该对象在下面讲解。

我们提供一个 **DataElement** 的接口，它的各个实现代表不同的值，这里提供两个实现：**RefElement** 和 **ValueElement**，前者表示一个 **ref** 的节点，后者表示一个 **value** 节点，**DataElement** 代码如下。

代码清单：code\IoC\main\org\crazyit\ioc\xml\construct\DataElement.java

```
public interface DataElement {  
    //返回数据的类型(ref 或者 value)  
    String getType();  
    //返回数据的值  
    Object getValue();  
}
```

代码清单：code\IoC\main\org\crazyit\ioc\xml\construct\RefElement.java

```
public class RefElement implements DataElement {  
    private Object value;  
    public RefElement(Object value) {  
        this.value = value;  
    }  
    public String getType() {  
        return "ref";  
    }  
    public Object getValue() {  
        return this.value;  
    }  
}
```

**ValueElement** 的实现与 **RefElement** 一样，在这里不写出代码。在接口中，另外还有 **getPropertyValue** 方法，通过封装一个 **PropertyElement** 来表示值的类型。**PropertyElement** 里面封装了一个 **DataElement** 对象，由于 **property** 节点下面可以有 **ref** 节点，也可以有 **value** 节点，**property** 节点下面除了可以有这两个节点之外，**property** 节点还有一个 **name** 的属性，因此再另外封装成一个 **PropertyElement**。

代码清单: code\IoC\main\org\crazyit\ioc\xml\property\property\PropertyElement.java

```
public class PropertyElement {
    //property 元素的 name 属性值
    private String name;
    //property 元素下的 ref 或者 value 属性对象
    private DataElement dataElement;
    public PropertyElement(String name, DataElement dataElement) {
        this.name = name;
        this.dataElement = dataElement;
    }
    //省略 name 和 dataElement 的 setter 和 getter 方法
}
```

#### 14.4.4 实现isLazy方法

接下来,我们去实现 ElementReader 接口的方法。新建实现类 ElementReaderImpl,下面实现 isLazy 方法。

代码清单: code\IoC\main\org\crazyit\ioc\xml\ElementReaderImpl.java

```
public boolean isLazy(Element element) {
    //获得 lazy-init 属性值
    String lazy = getAttribute(element, "lazy-init");
    Element parent = element.getParent();
    //获得父节点 beans 的 default-lazy-init 属性值
    Boolean parentLazy = new Boolean(getAttribute(parent, "default-lazy-init"));
    if (parentLazy) {
        //beans 节点需要延迟加载
        if ("false".equals(lazy)) return false;
        return true;
    } else {
        //根节点不需要延迟加载
        if ("true".equals(lazy)) return true;
        return false;
    }
}
```

isLazy 方法,判断一个节点是否需要延迟加载的话,首先要判断 beans 节点的 default-lazy-init 的值,再去判断本节点(bean)是否需要延迟加载。

#### 14.4.5 实现getConstructorElements方法

ElementReaderImpl 的 getConstructorElements 方法。

代码清单: code\IoC\main\org\crazyit\ioc\xml\ElementReaderImpl.java

```
public List<Element> getConstructorElements(Element element) {
    //得到 bean 节点下所有节点
    List<Element> children = element.elements();
    List<Element> result = new ArrayList<Element>();
    for (Element e : children) {
        //如果是 constructor-arg 节点, 加到结果集合中
        if ("constructor-arg".equals(e.getName())) {
```



```

        result.add(e);
    }
}
return result;
}

```

`getConstructorElements` 方法的实现十分简单，只是获得 `constructor-arg` 节点，加到结果集中返回即可，并不需要作其他处理。`getPropertyElements` 与以上的方法相同，只是 `getPropertyElements` 方法返回的是 `property` 节点集合。

#### 14.4.6 实现 `getAttribute` 和 `isSingleton` 方法

`ElementReaderImpl` 的 `getAttribute` 方法是返回一个节点的属性值，可以适用于任何的节点。`isSingleton` 判断一个 `bean` 节点是否为单态。

`ElementReaderImpl` 的 `getAttribute` 方法。

代码清单：code\IoC\main\org\crazyit\ioc\xml\ElementReaderImpl.java

```

public String getAttribute(Element element, String name) {
    String value = element.attributeValue(name);
    return value;
}

```

实现 `isSingleton` 方法。

代码清单：code\IoC\main\org\crazyit\ioc\xml\ElementReaderImpl.java

```

public boolean isSingleton(Element element) {
    Boolean singleton = new Boolean(getAttribute(element, "singleton"));
    return singleton;
}

```

这两个方法的实现不需要作任何的判断，只需要得到属性返回即可，由于我们在 `DTD` 文件中定义了 `singleton` 属性值只能是 `true` 或者 `false`，因此在这里不需要作处理。

#### 14.4.7 实现 `getAutowire` 方法

`getAutowire` 方法的实现与 14.4.4 中的 `isLazy` 方法一致，只是 `getAutowire` 方法返回的是一个 `Autowire` 对象，由于 `bean` 的延迟加载只有两种状态：延迟加载和不延迟加载，而 `autowire`（自动装配）有状态可能有多种，我们在本例中只提供了 `no` 和 `byName` 两种，如果以后还需要提供 `byType` 等状态时，我们可以加多一个 `Autowire` 的实现类。以下是 `getAutowire` 的实现。

代码清单：code\IoC\main\org\crazyit\ioc\xml\ElementReaderImpl.java

```

public Autowire getAutowire(Element element) {
    String value = this.getAttribute(element, "autowire");
    String parentValue = this.getAttribute(element.getParent(), "default-autowire");
    if ("no".equals(parentValue)) {
        //根节点不需要自动装配
        if ("byName".equals(value)) return new ByNameAutowire(value);
        return new NoAutowire(value);
    } else if ("byName".equals(parentValue)) {
        //根节点 byName
        if ("no".equals(value)) return new NoAutowire(value);
        return new ByNameAutowire(value);
    }
    return new NoAutowire(value);
}

```

```
}
```

以上的代码通过判断自动装配的形式，返回对应的 **Autowire** 对象。

#### 14.4.8 实现 `getConstructorValue` 和 `getPropertyValue` 方法

我们先实现 `getConstructorValue` 方法，该方法返回 **DataElement** 的集合，实现相对麻烦，如果 `constructor-arg` 下面是 `value`，那么要取得 `value` 节点的 `type` 属性值，再判断是哪一种基础数据类型，再对 `value` 节点的值进行转换。以下是 `getConstructorValue` 的实现。

代码清单：code\IoC\main\org\crazyit\ioc\xml\ElementReaderImpl.java

```
public List<DataElement> getConstructorValue(Element element) {
    //调用本类中的 getConstructorElements 方法取得全部的 constructor-arg 元素
    List<Element> cons = getConstructorElements(element);
    List<DataElement> result = new ArrayList<DataElement>();
    for (Element e : cons) {
        //获得 constructor-arg 下的 ref 元素或者 value 元素(只有一个)
        List<Element> els = e.elements();
        //将 Element 封装成 DataElement 对象
        DataElement dataElement = getDataElement(els.get(0));
        result.add(dataElement);
    }
    return result;
}

//判断 Element 类型是 ref 还是 value, 并将其封装成 DataElement 对象
private DataElement getDataElement(Element dataElement) {
    String name = dataElement.getName();
    if ("value".equals(name)) {
        String className = dataElement.attributeValue("type");
        String data = dataElement.getText();
        return new ValueElement(getValue(className, data));
    } else if ("ref".equals(name)) {
        return new RefElement(this.getAttribute(dataElement, "bean"));
    }
    return null;
}

//判断 className 的类型是否为基本类型, 是的话进行转换
private Object getValue(String className, String data) {
    if (isType(className, "Integer")) {
        return Integer.parseInt(data);
    } //省略 Boolean, Long, Short, Double, Float, Character 和 Byte 的转换
    else {
        return data;
    }
}

private boolean isType(String className, String type) {
    if (className.indexOf(type) != -1) return true;
    return false;
}
```

以上代码中的 `getDataElement` 方法，用来判断节点类型，是 `value` 还是 `ref`，并对它们的值进行封装，最后以 **DataElement** 返回。`getValue` 方法通过在 `value` 中定义的类型，来判断 `value` 节点的值，再

对这些值进行转换,至于为什么需要这样转换,我们将下面的章节中详细介绍。以下是 `getPropertyValue` 方法的实现,该方法主要获得 `property` 节点下面的 `value` 节点和 `ref` 节点的值。

代码清单: `code\IoC\main\org\crazyit\ioc\xml\ElementReaderImpl.java`

```
public List<PropertyElement> getPropertyValue(Element element) {
    //调用本类中的 getPropertyElements 取得所有的 property 节点
    List<Element> properties = getPropertyElements(element);
    List<PropertyElement> result = new ArrayList<PropertyElement>();
    for (Element e : properties) {
        //获得 property 下的 ref 元素或者 value 元素(只有一个)
        List<Element> els = e.elements();
        //将节点进行封装
        DataElement dataElement = getDataElement(els.get(0));
        //得到 property 节点的 name 属性
        String propertyNameAtt = this.getAttribute(e, "name");
        //将数据值和 property 元素的 name 属性封装成 PropertyElement 对象
        PropertyElement pe = new PropertyElement(propertyNameAtt, dataElement);
        result.add(pe);
    }
    return result;
}
```

`getPropertyValue` 方法与 `getConstructorValue` 方法的实现类似, `getConstructorValue` 是得到 `constructor-arg` 节点下的 `value` 节点值与 `ref` 节点值, `getPropertyValue` 方法同样也是得到 `value` 节点与 `ref` 节点的值。但是 `getPropertyValue` 方法返回的是 `PropertyElement` 对象集合, `PropertyElement` 对象中包含了 `DataElement` 和一个名字,这是由于 `property` 节点中还需要有一个 `name` 属性。

到了这里,解析 `Element` 的方法都已经实现了,但是这些方法会有什么用呢,在实现这些方法的时候,有些代码为什么需要这样去实现,我们留到下面的章节为大家详细的说明。

## 14.5 使用构造注入创建实例

前面的章节,并没有涉及如何去创建 `bean` 的实例,在本小节,我们将详细讲解如何使用 `bean` 的 `constructor-arg` 节点,去创建一个 `bean` 的实例。我们的配置文件中, `bean` 节点下面可以出现多个 `constructor-arg` 的节点,表示一个 `bean` 可以有多个构造参数,那么在创建 `bean` 的时候,就可以根据这些参数去创建实例,这就是依赖注入中的构造注入。

### 14.5.1 构造注入简介

构造注入,顾名思义,就是通过构造器将对应的值(`bean`)设置到目标对象中。在 `XML` 中只需要为 `bean` 节点配置子节点 `constructor-arg`,并分配相应的值即可, `IoC` 容器得到这些值后,就通过调用这个 `bean` 对应的 `Class` 的构造器创建该 `bean` 实例。

### 14.5.2 使用无参数构造器创建实例

使用构造器创建 `bean`,我们只需要得到该 `bean` 对应的 `Element` 对象,并得到 `bean` 的 `class` 属性,就可以通过反射来创建 `bean` 的实例。分两种情况,一种是没有构造参数的情况,另外一种配置了参数的情况。第一种没有构造参数的情况比较简单,我们先创建一个 `BeanCreator` 的接口,定义接口方法。

代码清单: `code\IoC\main\org\crazyit\ioc\context\BeanCreator.java`

```
public interface BeanCreator {
    //使用无参的构造器创建 bean 实例, 不设置任何属性
    Object createBeanUseDefaultConstruct(String className);
    //使用有参数的构造器创建 bean 实例, 不设置任何属性
    Object createBeanUseDefineConstruce(String className, List<Object> args);
}
```

**BeanCreator** 接口只有两个方法, 一个是使用默认的构造器创建对象, 参数为该对象对应的类名, 第二个方法是使用带参数的构造器创建对象, 第一个参数是该对象的类名, 第二个参数是构造器的参数集合, 我们的程序需要根据构造器的参数集合, 去寻找对应的构造器, 如果寻找不到, 我们可以抛出运行时异常。我们先实现第一个无参数构造器创建 **bean** 的方法, 该方法实现比较简单。

代码清单: code\IoC\main\org\crazyit\ioc\context\BeanCreatorImpl.java

```
public Object createBeanUseDefaultConstruct(String className) {
    try {
        //使用 Class.forName 通过类名创建 Class 对象
        Class clazz = Class.forName(className);
        //使用 newInstance 方法创建类的实例
        return clazz.newInstance();
    } catch (ClassNotFoundException e) {
        throw new BeanCreateException("class not found " + e.getMessage());
    } catch (Exception e) {
        throw new BeanCreateException(e.getMessage());
    }
}
```

以上简单的两行代码, 就可以调用无参数的构造器创建实例, 如果参数类名有错或者寻找不到该类, 那么将抛出 **ClassNotFoundException**, 我们将这个异常进行封装, 封成我们的自定义异常。那么外界调用这个方法的时候, 就直接给出 **bean** 节点的 **class** 属性值, 传递给该方法就可以返回对应的实例, 但是前提是需要判断该 **bean** 是否有构造参数。

### 14.5.3 使用有参数的构造器创建实例

13.5.2 中, 我们编写调用类的无参数构造器创建实例, 如果在 XML 配置中, **bean** 节点下面有 **constructor-arg** 节点, 那么我们就不会调用以上的方法创建实例, 下面实现 **BeanCreator** 接口的另外一个方法 **createBeanUseDefineConstruce**, 该方法根据参数集合的类型来决定我们需要使用哪个构造器, 当然, 构造器参数是有顺序的, 因此我们使用 **List** 集合。以下是 **createBeanUseDefineConstruce** 的实现。

代码清单: code\IoC\main\org\crazyit\ioc\context\BeanCreatorImpl.java

```
public Object createBeanUseDefineConstruce(String className,
    List<Object> args) {
    Class[] argsClass = getArgsClasses(args);
    try {
        Class clazz = Class.forName(className);
        Constructor constructor = findConstructor(clazz, argsClass);
        return constructor.newInstance(args.toArray());
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
        throw new BeanCreateException("class not found " + e.getMessage());
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
        throw new BeanCreateException("no such constructor " + e.getMessage());
    }
}
```

```

        } catch (Exception e) {
            e.printStackTrace();
            throw new BeanCreateException(e.getMessage());
        }
    }
    //获得构造器, 如果没有找到, 不抛出 NoSuchMethodException, 返回 null
    private Constructor getConstructor(Class clazz, Class[] argsClass) {
        try {
            Constructor constructor = clazz.getConstructor(argsClass);
            return constructor;
        } catch (NoSuchMethodException e) {
            return null;
        }
    }
    //根据类型和参数类型查找构造器
    private Constructor findConstructor(Class clazz, Class[] argsClass)
        throws NoSuchMethodException {
        Constructor constructor = getConstructor(clazz, argsClass);
        if (constructor == null) {
            Constructor[] constructors = clazz.getConstructors();
            for (Constructor c : constructors) {
                Class[] constructorArgsClass = c.getParameterTypes();
                //参数数量与构造器的参数数量相同
                if (constructorArgsClass.length == argsClass.length) {
                    if (isSameArgs(argsClass, constructorArgsClass)) {
                        return c;
                    }
                }
            }
        } else {
            return constructor;
        }
        throw new NoSuchMethodException("could not find any constructor");
    }
    //判断两个 Class 数组的类型是否一致
    private boolean isSameArgs(Class[] argsClass, Class[] constructorArgsClass) {
        for (int i = 0; i < argsClass.length; i++) {
            try {
                //将参数类型与构造器中的参数类型进行强制转换
                argsClass[i].asSubclass(constructorArgsClass[i]);
                //循环到最后一个都没有出错, 表示该构造器合适
                if (i == (argsClass.length - 1)) {
                    return true;
                }
            } catch (Exception e) {
                //有一个参数类型不符合, 跳出该循环
                break;
            }
        }
        return false;
    }

```

```
}  
//获得一个 Object 的 class, 主要判断是否基础类型  
private Class getClass(Object obj) {  
    if (obj instanceof Integer) {  
        return Integer.TYPE;  
    } else if (obj instanceof Boolean) {  
        return Boolean.TYPE;  
    } else if (obj instanceof Long) {  
        return Long.TYPE;  
    } else if (obj instanceof Short) {  
        return Short.TYPE;  
    } else if (obj instanceof Double) {  
        return Double.TYPE;  
    } else if (obj instanceof Float) {  
        return Float.TYPE;  
    } else if (obj instanceof Character) {  
        return Character.TYPE;  
    } else if (obj instanceof Byte) {  
        return Byte.TYPE;  
    }  
    return obj.getClass();  
}  
//得到参数的类型  
private Class[] getArgsClasses(List<Object> args) {  
    List<Class> result = new ArrayList<Class>();  
    for (Object arg : args) {  
        result.add(getClass(arg));  
    }  
    Class[] a = new Class[result.size()];  
    return result.toArray(a);  
}
```

代码说明:

- ❑ **getConstructor** 方法: 该方法主要根据类型和参数数组得到构造器, 如果查找不到, 则返回 `null`, 注意这里查找不到构造器有两种情况, 第一种是参数的类型不匹配, 第二种是参数错误。其中第一种情况较为复杂, 可能出现类的构造器中的参数是某一个接口, 而传入的参数是该接口的其中一个实现, 那么这样就会抛出 `NoSuchMethodException`, 我们这里直接返回 `null`。
- ❑ **getClass** 方法: 主要用于判断是否基础类型数据, 如果是的话, 返回对应的类型。
- ❑ **getArgsClasses** 方法: 该方法的参数是集合, 我们将这些集合对象转换成数组, 并调用 **getClass** 方法获得这些对象对应的类型。
- ❑ **isSameArgs** 方法: 判断参数中第一个类型数组与第二个类型数组里面对应的类型是否相同, 这里所指的类型是, 如果一个类是一个接口的实现类, 那么它们的类型相同, 该方法遍历所有数组中的类型, 如果全部都是类型相同, 那么就返回 `true`。
- ❑ **findConstructor** 方法: 参数为一个 `Class`, 一个参数值的类型数组, 先调用 **getConstructor** 方法取得构造器, 发生了参数值是对象中的构造参数的实现类, 那么 **getConstructor** 方法将返回 `null`, 我们再通过 `Class` 得到它所有的构造器, 遍历该 `Class` 所拥有的所有构造器, 先判断类构造器的参数数量与参数值的数量是否一致, 再判断它们 (类构造器参数和参数值) 是否都匹配, 调用 **isSameArgs** 方法判断, 如果可以找到对应的构造器, 就直接返回, 否则抛出

NoSuchMethodException。

最后接口方法 `createBeanUseDefineConstruce` 就可以调用 `findConstructor` 方法得到相应的构造器，得到构造器后，通过 `Constructor` 对象的 `newInstance` 方法创建实例。

注：在以上的代码中，直接通过 `Class` 的 `getConstructor` 方法虽然能找到构造器，但是如果我们使用了多态的特性，那么将不能直接找到该构造器，还要经过 `Class` 的 `asSubclass` 方法进行类型转换，如果转换成功，则说明是同一类型。

我们可以编写测试代码对对上所编写的程序进行测试，测试代码具体如下：

```
//无参数构造器创建实例, BeanCreatorObject1 是一个普通的 Java 对象, 只有 name 和 value 属性
String className = "org.crazyit.ioc.context.object.BeanCreatorObject1";
BeanCreatorObject1 obj = (BeanCreatorObject1)
    creator.createBeanUseDefaultConstruct(className);
System.out.println(obj);
System.out.println(obj.getName());
System.out.println(obj.getValue());
```

可以看到如下结果：

```
org.crazyit.ioc.context.object.BeanCreatorObject1 @1a73d3c
null
null
```

可以看到 `BeanCreatorObject1` 对象的 `name` 和 `value` 属性均为 `null`，以下再测试带参数的构造器：

```
//有参数构造器创建实例
String className = "org.crazyit.ioc.context.object.BeanCreatorObject2";
//构造参数集合
List<Object> args = new ArrayList<Object>();
args.add("yangenxiong");
args.add("crazyit");
BeanCreatorObject2 obj = (BeanCreatorObject2)
    creator.createBeanUseDefineConstruce(className, args);
System.out.println(obj.getName());
System.out.println(obj.getValue());
```

可以看到通过构造器注入两个属性，可以打印相关的值。到此，我们编写了创建 `bean` 实例的接口，如果需要使用这个接口的时候，可以直接将这两个接口方法所需要的参数传递给它们，就可以得到类的实例。

## 14.6 实现设值注入

设值注入，就是调用对象的 `setter` 方法，将配置好的值设入对象中，但是前提是该对象已经创建好了，在 14.5 中我们就已经编写了创建对象的方法，那么本小节我们实现设值注入。我们在定义 `DTD` 的时候，`bean` 节点下面可以有多个 `property` 节点，`property` 节点下面只允许有一个 `ref` 节点或者 `value` 节点，表示一个对象可以多次进行设值注入，但是每次注入只允许有一个值，可以是基础类型数据，也可以是自定义类型，但是前提是需要进行设值注入的对象必须提供 `setter` 方法，且该 `setter` 方法只允许有一个参数。

### 14.6.1 实现非自动装配的设值注入

我们在定义 `DTD` 的时候，就已经确定了一个 `bean` 如何进行自动装配，如果该 `bean` 不需要自动装配（`autowire` 属性值为 `no`）的话，那么就会根据 `bean` 节点下面所有的 `property` 节点的值，通过这些值

去 **bean** 对应的类型中去寻找 **setter** 方法，找到方法名称与方法参数都匹配的方法，再执行即可，下面实现不需要自动装配。

新建接口 **PropertyHandler**，为该接口提供一个 **setProperties** 的方法，参数为对象的实例和各个属性值的 **Map** 对象，**Map** 对象的 **key** 为 XML 中配置的 **property** 节点的 **name** 属性，**value** 为对应的值。

代码清单：code\IoC\main\org\crazyit\ioc\context\PropertyHandler.java

```
public interface PropertyHandler {
    //为对象 obj 设置属性
    Object setProperties(Object obj, Map<String, Object> properties);
}
```

该接口 **setProperties** 方法的返回是参数中的 **obj** 对象，返回的时候，该对象中的属性已经被设值。以下是该方法的实现。

代码清单：code\IoC\main\org\crazyit\ioc\context\PropertyHandlerImpl.java

```
public Object setProperties(Object obj, Map<String, Object> properties) {
    Class clazz = obj.getClass();
    try {
        for (String key : properties.keySet()) {
            String setterName = getSetterMethodName(key);
            Class argClass = getClass(properties.get(key));
            Method setterMethod = getSetterMethod(clazz, setterName, argClass);
            setterMethod.invoke(obj, properties.get(key));
        }
        return obj;
    } catch (NoSuchMethodException e) {
        throw new PropertyException("setter method not found " + e.getMessage());
    } catch (IllegalArgumentException e) {
        throw new PropertyException("wrong argument " + e.getMessage());
    } catch (Exception e) {
        throw new PropertyException(e.getMessage());
    }
}

//获得一个 Object 的 class
private Class getClass(Object obj) {
    if (obj instanceof Integer) {
        return Integer.TYPE;
    } //以下省略 Boolean, Long, Short, Double, Float, Character, Byte
    return obj.getClass();
}

//得到 setter 方法的 Method 对象
private Method getSetterMethod(Class objClass, String methodName,
    Class argClass) throws NoSuchMethodException {
    //使用原类型获得方法，如果没有找到该方法，则得到 null
    Method argClassMethod = getMethod(objClass, methodName, argClass);
    //如果找不到原类型的方法，则找该类型所实现的接口
    if (argClassMethod == null) {
        //得到所有名字为 methodName 的方法
        List<Method> methods = getMethods(objClass, methodName);
        Method method = findMethod(argClass, methods);
        if (method == null) {
            //找不到任何方法
            throw new NoSuchMethodException(methodName);
        }
    }
}
```



```

        }
        return method;
    } else {
        return argClassMethod;
    }
}
//得到所有名字为 methodName 并且只有一个参数的方法
private List<Method> getMethods(Class objClass, String methodName) {
    List<Method> result = new ArrayList<Method>();
    for (Method m : objClass.getMethods()) {
        if (m.getName().equals(methodName)) {
            //得到方法的所有参数, 如果只有一个参数, 则添加到集合中
            Class[] c = m.getParameterTypes();
            if (c.length == 1) {
                result.add(m);
            }
        }
    }
    return result;
}
//遍历所有的方法, 判断方法中的参数与 argClass 是否是同一类型
private Method findMethod(Class argClass, List<Method> methods) {
    for (Method m : methods) {
        //判断参数类型与方法参数类型是否一致
        if (isMethodArgs(m, argClass)) {
            return m;
        }
    }
    return null;
}
//判断一个方法的参数类型是否与 argClass 类型一样, 有可能 argClass 是方法参数类型的实现类
private boolean isMethodArgs(Method m, Class argClass) {
    //得到方法的所有参数类型
    Class[] c = m.getParameterTypes();
    //如果方法的参数不是 1 个, 则表示该 Method 不是我们所需要的 setter 方法
    if (c.length == 1) {
        try {
            //将参数类型(argClass)与方法中的参数类型进行强制转换, 不抛异常返回该 Method
            argClass.asSubclass(c[0]);
            return true;
        } catch (ClassCastException e) {
            return false;
        }
    }
    return false;
}
//将参数 s 的首字母变为大写
private String upperCaseFirstWord(String s) {
    String firstWord = s.substring(0, 1);
    String upperCaseWord = firstWord.toUpperCase();

```

```

        return s.replaceFirst(firstWord, upperCaseWord);
    }
    //返回一个属性的 setter 方法
    private String getSetterMethodName(String propertyName) {
        return "set" + upperCaseFirstWord(propertyName);
    }
    //根据方法名和参数类型得到方法, 如果没有该方法返回 null
    private Method getMethod(Class objClass, String methodName, Class argClass) {
        try {
            Method method = objClass.getMethod(methodName, argClass);
            return method;
        } catch (NoSuchMethodException e) {
            return null;
        }
    }
}

```

方法说明:

- ❑ **upperCaseFirstWord** 方法: 将参数中的字符串的首字母变为大写, 当我们得到一个属性名为 **age** 配置属性 (`<property name="age">`) 后, 我们需要得到 **age** 的 **setter** 方法 (**setAge**), 因此需要将 **age** 的首字母变成大写。
- ❑ **getSetterMethodName** 方法: 得到 **setter** 方法名, 例如有 **age** 属性, 返回 **setAge** 字符串。
- ❑ **isMethodArgs** 方法: 该方法有两个参数, 第一个参数为 **Method** 对象, 第二个参数为 **Class** 对象, 判断该 **Class** 是否与 **Method** 的参数 **Class** 一样, 与构造注入的情况一样, 当 **Method** 的参数类型是一个接口, 而具体的参数值是该接口的实现类, 我们就需要使用 **Class** 中的 **asSubclass** 方法进行强制转换, 成功转换, 则表示该 **Method** 对象就是我们所需要的 **setter** 方法。
- ❑ **getMethods** 方法: 给一个类型和方法名, 到该类型中寻找方法名与参数的方法名一致的方法, 并封装成集合返回, 由于我们可以在类中可以有多个名字相同的方法, 只要这些方法的参数数量或者类型不一样, 就可以存在多个, 因此在一个类中有相同方法名的方法可能有多个。
- ❑ **findMethod** 方法: 在方法的集合遍历每个方法, 判断方法的参数类型是否与 **argClass** 一致, 判断类型是否一致使用以上的 **isMethodArgs** 进行判断。如果类型一致, 则马上返回。
- ❑ **getMethod** 方法: 直接通过类名、方法名和方法参数类型查找方法, 如果查找不到, 则直接返回 **null**。
- ❑ **getSetterMethod** 方法: 根据类型、方法名和方法参数值类型得到对应的 **setter** 方法。先调用 **getMethod** 方法寻找方法, 如果该方法的参数类型是接口, 而参数值类型是该接口的实现类, 那么将和构造注入一样, 找不到对应的方法。因此在 **getSetterMethod** 方法中, 再调用 **findMethod** 方法进行查找。
- ❑ **getClass** 方法: 与构造注入中的 **getClass** 实现一致, 这里不详细描述。

实现了以上的工具方法后, 就可以实现接口方法, 接口方法 **setProperties**, 通过一系列的工具体方法, 先得到对象的类型, 再得到所有遍历所有在 **bean** 节点下配置的 **property** 节点及其对应的值 (**setProperties** 方法的第二个参数), 遍历这些参数, 得到 **setter** 方法的字符串名字, 再得到方法参数的类型 (只有一个参数), 最后得后 **setter** 方法的 **Method** 对象, 最后执行这个 **Method**, 那么就等于执行了对象的 **setter** 方法了。

以下编写一些测试代码来检查上面的代码是否正确:

```

//PropertyHandlerObject1 是普通的 Java 对象
PropertyHandlerObject1 obj = new PropertyHandlerObject1();

```

```

Map<String, Object> properties = new HashMap<String, Object>();
properties.put("name", "yangenxiong");
properties.put("age", 10);
//School 是普通的 Java 对象
School school = new School();
properties.put("school", school);
//调用 PropertyHandler 的 setProperties 方法得到新的对象
PropertyHandlerObject1 newObj = (PropertyHandlerObject1)
handler.setProperties(obj, properties);
//打印结果
System.out.println(newObj.getName());
System.out.println(newObj.getAge());
System.out.println(newObj.getSchool());
System.out.println(school);

```

我们可以看到结果如下：

```

yangenxiong
10
org.crazyit.ioc.context.object.School@1f20eeb
org.crazyit.ioc.context.object.School@1f20eeb

```

最后两行的打印，PropertyHandlerObject1 对象中的 School 对象和我们所创建（new）的 School 是同一对象，可以证明已经通过 setter 方法将值设入到 PropertyHandlerObject1 的实例中。在以上的测试代码中，PropertyHandlerObject1 需要为 name、age 和 school 提供 setter 方法，方法名分别是 setName、setAge 和 setSchool。

讲到这里，我们已经明白了什么是设值注入，无非是将一些值通过对象的 setter 方法设值到对象中，而这些需要被设置的值和目标对象，都是通过配置文件来读取的。

## 14.6.2 实现根据名字自动装配

我们在定义 DTD 的时候，就声明了一个 autowire 的属性，当 autowire 属性值为 no 的时候，就不使用自动装配，该属性还有另外的值：byName，通过名字自动装配，那么本小节将介绍如何通过名字进行自动装配。

14.6.1 中不自动装配，是得到 bean 节点下面的所有 property 节点的值，再查找对应的 setter 方法进行调用。byName 自动装配恰好相反，找到对象中的所有 setter 方法，对 setter 方法的名称作处理，得到属性名，例如得到 setSchool 方法，那么就去 IoC 容器中寻找名字为 school 的 bean，如果可以找到，则调用 setter 方法进行设值，没有找到就忽略，这样就可以实现自动装配，不再需要进行任何的 property 配置。这样做的好处在于，可以简化配置文件，但是同时将程序的耦合降到代码程度，我们提供配置文件，是为了让程序的耦合只在配置文件中体现，是否进行 autowire 各有利弊。

在本小子中，我们只需要编写查找所有 setter 方法和执行 setter 方法的程序即可，由于我们尚未开始编写 IoC 容器的主体代码，我们本小节所编写的代码，将在 14.7 中详细描述用处。

在 PropertyHandler 中加入两个接口方法。

代码清单：code\IoC\main\org\crazyit\ioc\context\PropertyHandler.java

```

//返回一个对象里面所有的 setter 方法，封装成 map, key 为 setter 方法名不要 set
Map<String, Method> getSetterMethodsMap(Object obj);
/**
 * 执行一个方法
 * @param object 需要执行方法的对象
 * @param argBean 参数的 bean
 * @param method 方法对象

```

```
*/
void executeMethod(Object object, Object argBean, Method method);
```

注意 `getSetterMethodsMap` 方法, 由于我们需要得到所有的 `setter` 方法, 因此就需要使用一个 `Map` 来保存, `Map` 的 `key` 为属性名称, 在 `IoC` 的主体代码中, 我们将以这个 `key` 为 `bean` 的 `id`, 去容器中查找对应的 `bean`, `Map` 的 `value` 为一个方法对象, 当在容器中找到对应的 `bean` 时, 就调用 `PropertyHandler` 接口中的 `executeMethod` 方法执行 `setter` 方法。

以下是这两个方法的实现。

代码清单: `code\IoC\main\org\crazyit\ioc\context\PropertyHandlerImpl.java`

```
public void executeMethod(Object object, Object argBean, Method method) {
    try {
        //获取方法的参数类型
        Class[] parameterTypes = method.getParameterTypes();
        //如果参数数量不为 1, 则不执行该方法
        if (parameterTypes.length == 1) {
            //调用 isMethodArgs 方法判断参数类型是否一致
            if (isMethodArgs(method, parameterTypes[0])) {
                method.invoke(object, argBean);
            }
        }
    } catch (Exception e) {
        throw new BeanCreateException("autowire exception " + e.getMessage());
    }
}

public Map<String, Method> getSetterMethodsMap(Object obj) {
    List<Method> methods = getSetterMethodsList(obj);
    Map<String, Method> result = new HashMap<String, Method>();
    for (Method m : methods) {
        String propertyName = getMethodNameWithoutSet(m.getName());
        result.put(propertyName, m);
    }
    return result;
}

//获得所有的 setter 方法
private List<Method> getSetterMethodsList(Object obj) {
    Class clazz = obj.getClass();
    Method[] methods = clazz.getMethods();
    List<Method> result = new ArrayList<Method>();
    for (Method m : methods) {
        if (m.getName().startsWith("set")) {
            result.add(m);
        }
    }
    return result;
}

//将 setter 方法还原, 如 setName 作为参数, 最后返回 name
private String getMethodNameWithoutSet(String methodName) {
    String propertyName = methodName.replaceFirst("set", "");
    String firstWord = propertyName.substring(0, 1);
    String lowerFirstWord = firstWord.toLowerCase();
    return propertyName.replaceFirst(firstWord, lowerFirstWord);
}
```

```
}
```

代码说明：

- ❑ `getMethodNameWithoutSet` 方法：将方法名的 `set` 去掉，并将首字母变回小写，例如 `setAge` 方法，最后返回 `age` 字符串。
- ❑ `getSetterMethodsList` 方法：得到对象中所有方法名以 `set` 开头的方法对象集合。
- ❑ `isMethodArgs` 方法：以上代码的黑体部分为 `isMethodArgs` 方法，重用了 14.6.1 中的 `isMethodArgs` 方法。

以上代码中的 `executeMethod` 方法，得到 `Method` 对象的所有参数类型后，再判断这些参数数量是否等于 1，再判断参数的类型是否一致，最后就调用 `Method` 的 `invoke` 方法。下面编写一些测试来验证以上的代码：

```
//测试获得全部的 setter 方法
PropertyHandlerObject1 obj = new PropertyHandlerObject1();
Map<String, Method> result = handler.getSetterMethodsMap(obj);
System.out.println(result.get("name"));
System.out.println(result.get("age"));
System.out.println(result.get("school"));
```

打印结果如下：

```
public void org.crazyit.ioc.context.object.PropertyHandlerObject1.setName(java.lang.String)
public void org.crazyit.ioc.context.object.PropertyHandlerObject1.setAge(int)
public void org.crazyit.ioc.context.object.PropertyHandlerObject1.setSchool(org.crazyit.ioc.context.object.School)
```

以下代码测试 `executeMethod` 方法：

```
PropertyHandlerObject1 obj = new PropertyHandlerObject1();
School school = new School();
Method m = obj.getClass().getMethod("setSchool", School.class);
handler.executeMethod(obj, school, m);
System.out.println(obj.getSchool());
System.out.println(school);
```

得到结果如下：

```
org.crazyit.ioc.context.object.School@b179c3
org.crazyit.ioc.context.object.School@b179c3
```

我们在前面讲过，需要进行自动装配，先得到 `setter` 方法，再到 `IoC` 容器中得到这个 `setter` 方法所对应的名称，再根据这些名称到 `IoC` 容器中查找对应的 `bean`，最后再决定是否调用 `PropertyHandler` 的 `executeMethod` 方法（以上代码中的 `executeMethod` 方法），我们这里没有任何关于从 `IoC` 容器中查找这些 `bean` 的代码，只是更关注于值设置。

本节主要讲了如何实现非自动装配的设值注入，根据名字自动装配的设值注入，与 14.5 节中的构造注入，我们已经大致上实现了设值注入和构造注入，在下一节中，我们编写 `IoC` 容器的主体代码。

## 14.7 实现IoC容器

14.6 节中，我们已经将大部分的将会使用到的代码编写，那么接下来，就可以编写 `IoC` 容器的主体代码，实现 `IoC` 容器的顺序按照我们本章中所开始编码的顺序进行，先从 14.4 节开始读取 XML 文件，再使用 14.5 节中的创建实例的代码创建 `bean` 的实例，最后使用 14.6 的设值注入代码，为对象设置相应的值。

### 14.7.1 定义接口方法

我们需要创建一个 IoC 容器的对象，通过该对象可以直接从容器中得到相应的信息。以下新建 `ApplicationContext` 接口。

代码清单：code\IoC\main\org\crazyit\ioc\context\ApplicationContext.java

```
public interface ApplicationContext {  
    /**  
     * 根据 id 找到 bean  
     * @param id 配置文件中 bean 节点的 id  
     * @return 该 bean 的实例  
     */  
    Object getBean(String id);  
    /**  
     * IoC 容器中是否包含 id 为参数的 bean  
     * @param id  
     * @return true 容器中包含这个 bean  
     */  
    boolean containsBean(String id);  
    /**  
     * 判断一个 bean 是否为单态  
     * @param id 配置文件中 bean 节点的 id  
     * @return 如果该 bean 是单态的，返回 true  
     */  
    boolean isSingleton(String id);  
    /**  
     * 获得 bean，如果从容器中找不到该 bean，返回 null  
     * @param id 配置文件中 bean 节点的 id  
     * @return 找到的 bean 实例  
     */  
    Object getBeanIgnoreCreate(String id);  
}
```

`ApplicationContext` 接口代表一个 IoC 容器，根据这个接口，我们可以提供不同的实现，以下是该接口的方法说明：

- ❑ `getBean` 方法：根据 bean 的 id 在 IoC 容器中查找 bean 的实例，如果查找不到，则尝试进行创建该 bean 的实例。
- ❑ `containsBean` 方法：判断一个 id 是否在 IoC 容器中存在 bean。
- ❑ `isSingleton` 方法：判断一个 bean 的 id 所对应的 bean 是否为单态。
- ❑ `getBeanIgnoreCreate` 方法：直接从容器中查找 bean，如果查找不到，则返回 null，不进行创建。

定义了以上的方法后，我们就可以创建对应的实现类，逐步去实现以上的方法。以上的方法中，实现 `getBean` 方法逻辑相对较为复杂，我们在下面的章节慢慢讲解。

### 14.7.2 实现containsBean方法和isSingleton方法

这两个方法实现比较简单，因此我们放到一起讲述。上一小节定义了容器接口后，我们为它新建一个抽象类 `AbstractApplicationContext`，该抽象类需要实现 `ApplicationContext` 里的所有方法，那么可能会问，已经实现了所有的方法，为什么还要搞成抽象类呢？这是因为我们还需要编写其他的类去继承

这个抽象类，因此不允许外界直接去使用 **new** 关键字来创建这个类。

接口的几个方法，在 **AbstractApplicationContext** 中，我们暂时提供空实现，现在新建一个方法，用于处理读取的 XML 文件。

代码清单：code\IoC\main\org\crazyit\ioc\context\AbstractApplicationContext.java

```
public abstract class AbstractApplicationContext implements ApplicationContext {
    //元素加载对象
    protected ElementLoader elementLoader = new ElementLoaderImpl();
    //文档持有对象
    protected DocumentHolder documentHolder = new XmlDocumentHolder();
    //缓存的 beans
    protected Map<String, Object> beans = new HashMap<String, Object>();
    //属性处理类
    protected PropertyHandler propertyHandler = new PropertyHandlerImpl();
    //创建 bean 对象的接口
    protected BeanCreator beanCreator = new BeanCreatorImpl();
    //Element 元素读取类
    protected ElementReader elementReader = new ElementReaderImpl();
    //读取 xml 文件，将各个元素缓存
    protected void setUpElements(String[] xmlPaths) {
        URL classPathUrl = AbstractApplicationContext.class.getClassLoader().getResource(".");
        String classPath = java.net.URLDecoder.decode(classPathUrl.getPath(), "utf-8");
        for (String path : xmlPaths) {
            Document doc = documentHolder.getDocument(classPath + path);
            elementLoader.addElements(doc);
        }
    }
}
```

注意以上的各个类属性，这些都属性都是我们在前面的章节中准备的接口，包括读取 XML 的接口、解析 **Element** 的接口、创建类实例的接口和类属性的处理接口，我们在前面几节中已经对这些接口进行了实现，现在这些类终于可以派上用场了。

以上的 **setUpElement** 方法，参数是字符串数组，先得到环境变量的 URL，再与参数的字符串相加，得到的就是需要 IoC 容器去加载的配置文件的绝对路径了（需要将得到的 **classPath** 字符串进行解码）。得到配置文件的路径后，再通过我们在 14.4 中编写的 **DocumentHolder** 接口和 **ElementLoader** 接口，将各份文档中所对应的 **Document** 对象和这些 **Document** 对象里面的 **Element** 对象进行缓存，这里的 **Element** 对象，实际上就是每一个 **bean** 节点，我们在定义 DTD 的时候，就已经确定了 **beans** 节点下面只可以出现 **bean** 节点，出现次数为 0 次或者多次。得到所有的 **Document** 对象和所有的 **Element** 对象，那么实现 **containsBean** 方法和 **isSingleton** 方法就十分简单，以下为两个方法的实现。

代码清单：code\IoC\main\org\crazyit\ioc\context\AbstractApplicationContext.java

```
public boolean containsBean(String id) {
    //调用 ElementLoader 对象，根据 id 得到对应的 Element 对象
    Element e = elementLoader.getElement(id);
    return (e == null) ? false : true;
}

public boolean isSingleton(String id) {
    //使用 ElementLoader 方法获得对应的 Element
    Element e = elementLoader.getElement(id);
    //使用 ElementReader 判断是否为单态
    return elementReader.isSingleton(e);
}
```

`containsBean` 方法的实现，使用了 `ElementLoader` 接口，该接口中各个方法已经在本章中的 14.4.2 中有详细说明。`isSingleton` 方法除了使用 `ElementLoader` 接口外，还使用了 `ElementReader` 接口，该接口主要用于处理 `Element` 对象的解析，具体的使用在本章的 14.4.3 中的详细说明。

这里需要注意的是，`containsBean` 方法判断容器中是否存在名字为 `id` 的 `bean`，该方法的实现是从所有的配置文件的元素中寻找对应的 `bean`，而不是去实现缓存的 `Map`（`AbstractApplicationContext` 中的 `beans` 属性）中去寻找，这是由于我们容器会出现有一些 `bean` 是非单态的，也就是不会在 `beans` 属性中进行缓存，因此我们需要从所有的 `Element` 中查找。

`isSingleton` 方法与 `containsBean` 方法一样，同样都是去所有的 `Element` 中查找相应的对象，由于我们在 14.4.3 中已经为 `ElementReader` 接口实现了 `isSingleton` 方法，所以这里的代码就相对变得简单了。

### 14.7.3 实现 `getBean` 方法

`ApplicationContext` 的 `getBean` 方法是我们这个 IoC 的核心方法，通过一个 `bean` 的 `id`，去获取该 `bean` 的实例，外界调用这个方法请求 `bean` 实例的时候，我们就从 `AbstractApplicationContext` 的 `beans` 属性中查找 `bean` 实例，如果查找不到，那么就进行创建。创建的顺序是先根据构造器来创建实例，再对该 `bean` 进行设值。

实现 `getBean` 方法，我们先创建相应的实例，为了更好的理解，暂时不进行设值注入。

代码清单：code\IoC\main\org\crazyit\ioc\context\AbstractApplicationContext.java

```
public Object getBean(String id) {
    Object bean = this.beans.get(id);
    //如果获取不到该 bean，则创建
    if (bean == null) {
        //判断处理单态或者非单态的 bean
        bean = handleSingleton(id);
    }
    return bean;
}
//处理 bean，如果是单态的，则加到 map 中，非单态，则创建并返回
protected Object handleSingleton(String id) {
    Object bean = createBean(id);
    //调用本类中已经实现的 isSingleton 方法
    if (isSingleton(id)) {
        //单态的话，放到 map 中
        this.beans.put(id, bean);
    }
    return bean;
}
//创建一个 bean 实例并设置属性，如果找不到该 bean 对应的配置文件的 Element 对象，抛出异常
protected Object createBean(String id) {
    Element e = elementLoader.getElement(id);
    if (e == null) throw new BeanCreateException("element not found " + id);
    Object result = instance(e);
    System.out.println("创建 bean: " + id);
    System.out.println("该 bean 的对象是: " + result);
    //设值注入暂时不实现===
    return result;
}
//实例化一个 bean，如果该 bean 的配置有 constructor-arg 元素，那么使用带参数的构造器
```



```

protected Object instance(Element e) {
    String className = elementReader.getAttribute(e, "class");
    //得到 bean 节点下面的 constructor-arg 节点
    List<Element> constructorElements = elementReader.getConstructorElements(e);
    //判断使用什么构造器进行创建(判断标准为 bean 元素下是否有 constructor-arg 子元素)
    if (constructorElements.size() == 0) {
        //没有 constructor-arg 子元素, 使用无参构造器
        return beanCreator.createBeanUseDefaultConstruct(className);
    } else {
        //有 constructor-arg 子元素, 使用有参数构造器, 构造注入参数
        List<Object> args = getConstructArgs(e);
        return beanCreator.createBeanUseDefineConstruce(className, args);
    }
}
//得到一个 bean 里面配置的构造参数
protected List<Object> getConstructArgs(Element e) {
    List<DataElement> datas = elementReader.getConstructorValue(e);
    List<Object> result = new ArrayList<Object>();
    for (DataElement d : datas) {
        if (d instanceof ValueElement) {
            d = (ValueElement)d;
            result.add(d.getValue());
        } else if (d instanceof RefElement) {
            //如果是引用元素, 则直接调 getBean 去获取(获取不到则创建)
            d = (RefElement)d;
            String refId = (String)d.getValue();
            result.add(this.getBean(refId));
        }
    }
    return result;
}
}

```

方法说明:

- ❑ **getConstructArgs** 方法: 该方法得到 bean 节点下面所有 constructor-arg 节点的值, constructor-arg 节点的值可能是 ref 节点的 bean 属性值, 也可能是 value 节点的值, 通过调用 ElementReader 接口的 getConstructorValue 方法取得所有的值对象集合, getConstructorValue 方法已经帮我们将这些值封装成 DataElement 对象, 再判断 DataElement 的类型, 如果是 ValueElement, 就获得 ValueElement 的 value 属性, 添加到结果的集合中, 如果是 RefElement, 再次调用 getBean 方法得到容器中的 bean 实例, 实际上这里是递归调用。具体 getConstructorValue 方法的实现与使用, 请看 14.4.8 中该方法的详细描述。
- ❑ **instance** 方法: 通过一个 Element 元素得到该元素所对应的 bean 的实例, 这里使用了构造器创建 bean, 先取得 bean 的 class 属性值, 再获得 bean 下面所有的 constructor-arg 节点, 判断节点的数量, 如果 constructor-arg 节点数量为 0 的话, 那么就直接使用 BeanCreator 接口的 createBeanUseDefaultConstruct 方法, 调用无参数构造器创建实例。如果 constructor-arg 节点参数不为 0, 就调用本类中的 getConstructArgs 方法获得参数, 再调用 BeanCreator 的 createBeanUseDefineConstruce 方法调用对应的构造器创建实例。BeanCreator 接口的实现与使用, 请参看本章 14.5 节。
- ❑ **createBean** 方法: 该方法根据参数的 bean 的 id, 先得到对应的 Element 对象, 如果找不到该

Element 对象，则抛出异常，得到该对象后，调用本类的 instance 方法创建 bean 的实例，再使用设值注入为该 bean 设置属性，设置属性我们这里暂时不提供实现，直接返回该 bean 的实例。

- ❑ handleSingleton 方法：先调用 createBean 方法创建该 bean 的实例后，再判断这个 bean 是否被设置为单态的，如果被设置为单态的（bean 节点的 singleton 属性为 true），那么就将对象放到 Map 中（本类的 beans 属性）进行保存，如果非单态的，那么就直接返回该 bean 实例。

接口方法 getBean 的实现，先从 AbstractApplicationContext 的 beans 属性中查找 bean 实例，如果查不到，就调用 createBean 方法创建，最后返回该实例。

以上的 getBean 方法的实现只是得到一个实例，该实例中的属性我们还没有设置，现在可以实现设值注入，为已经创建的对象设置相应的属性。在实现前我们需要明白，本章中设值注入提供两种方式，一种是通过名字自动装配，另外一种是不自动装配，因此就需要进行判断。以上是对 AbstractApplicationContext 中的 createBean 方法的补充。

代码清单：code\IoC\main\org\crazyit\ioc\context\AbstractApplicationContext.java

```
//创建一个 bean 实例并设置属性，如果找不到该 bean 对应的配置文件的 Element 对象，抛出异常
protected Object createBean(String id) {
    Element e = elementLoader.getElement(id);
    if (e == null) throw new BeanCreateException("element not found " + id);
    Object result = instance(e);
    System.out.println("创建 bean: " + id);
    System.out.println("该 bean 的对象是: " + result);
    //设值注入，先判断是否自动装配
    Autowire autowire = elementReader.getAutowire(e);
    if (autowire instanceof ByNameAutowire) {
        //使用名称自动装配
        autowireByName(result);
    } else if (autowire instanceof NoAutowire) {
        //不自动装配，通过<property>属性
        setterInject(result, e);
    }
    return result;
}

/**
 * 自动装配一个对象，得到该 bean 的所有 setter 方法，再从容器中查找对应的 bean
 * 例如，如果 bean 中有一个 setSchool(School)方法，那么就去查名字为 school 的 bean，
 * 再调用 setSchool 方法设入对象中
 * @param obj
 */
protected void autowireByName(Object obj) {
    //得到所有的 setter 方法
    Map<String, Method> methods = propertyHandler.getSetterMethodsMap(obj);
    //遍历所有的 setter 方法
    for (String s : methods.keySet()) {
        //得到对应的 bean 元素
        Element e = elementLoader.getElement(s);
        //没有对应的元素配置，继续循环
        if (e == null) continue;
        //调用 getBean 方法返回 bean
        Object bean = this.getBean(s);
    }
}
```

```

        //获得 Method 对象
        Method method = methods.get(s);
        //调用 PropertyHandler 的 executeMethod 方法执行 setter 方法
        propertyHandler.executeMethod(obj, bean, method);
    }
}
//通过 property 元素为参数 obj 设置属性
protected void setterInject(Object obj, Element e) {
    //得到 property 节点下面的值, 包括 ref 的 bean 属性和 value 的值
    List<PropertyElement> properties = elementReader.getPropertyValue(e);
    //调用本类的 getPropertyArgs 方法装参数集合重新封装成 Map
    Map<String, Object> propertiesMap = getPropertyArgs(properties);
    //将参数的 Map 对象和 bean 的实例, 调用 PropertyHandler 接口的 setProperties 方法设置
    propertyHandler.setProperties(obj, propertiesMap);
}
//以 map 的形式得到需要注入的参数对象, key 为 setter 方法名(不要 set), value 为参数对象
protected Map<String, Object> getPropertyArgs(List<PropertyElement> properties) {
    Map<String, Object> result = new HashMap<String, Object>();
    for (PropertyElement p : properties) {
        //从 PropertyElement 中得到 DataElement 对象
        DataElement de = p.getDataElement();
        if (de instanceof RefElement) {
            //从容器中得到 bean 的实例, 再设置入 map 中
            result.put(p.getName(), this.getBean((String)de.getValue()));
        } else if (de instanceof ValueElement) {
            //得到 ValueElement 的值
            result.put(p.getName(), de.getValue());
        }
    }
    return result;
}
}

```

方法说明:

- ❑ **getPropertyArgs** 方法: 当我们得到属性集合后 (已经封装成 **PropertyElement**), 就这些属性集合再次进行处理, 返回结果为 **Map** 对象, 由于参数的集合中存放的是 **PropertyElement**, 那么我们可以从 **PropertyElement** 中得到 **DataElement** 对象, 再判断该对象是哪一种类型, 如果是 **RefElement** (**property** 节点下面的 **ref** 节点), 就再次调用容器的 **getBean** 方法取得属性所对应的 **bean**。
- ❑ **setterInject** 方法: 得到一个对象的实例和对应的 **Element** 对象, 就根据 **Element** 对象得到配置中的所有 **property** 节点的值, 再调用本类中的 **getPropertyArgs** 方法将参数集合封装成 **Map** 对象, 最后将实例与参数的 **Map** 对象传给 **PropertyHandler** 接口的 **setProperties** 方法进行设置。**PropertyHandler** 接口的实现与使用, 请看本章的 14.6.1 节。
- ❑ **autowireByName** 方法: 根据名字进行自动装配, 得到一个 **bean** 的实例后, 再获得这个 **bean** 的所有的 **setter** 方法, 得到相应的名字后, 再去 **IoC** 容器中查找这个 **bean**, 找到相应的 **bean** 后, 再调用 **setter** 方法将 **bean** 设值到这个实例中。该方法中使用了 **PropertyHandler** 的 **getSetterMethodsMap** 方法和 **executeMethod** 方法, 具体的实现与使用, 在本章的 14.6.2 节中已经详细描述。

最后将 **createBean** 方法进行补充, 以上代码的黑体部分就是补充的实现, 得到实例后, 先判断自

动装配的类型，如果是根据名字自动装配（byName），则调用本类的 `autowireByName` 方法，如果不需要自动装配，得调用 `setterInject` 方法。在以上代码中的 `getPropertyArgs` 方法，如果 `property` 节点下面有 `ref` 节点，那么就再次调用 `getBean` 方法得到 `bean`，实际上这里是一个递归调用，与 `constructor-arg` 节点的 `ref` 节点类似（`AbstractApplicationContext` 类中的 `getConstructArgs` 方法）。

`getBean` 方法已经实现，那么只剩下一个 `getBeanIgnoreCreate` 的接口方法没有实现了，该方法十分简单，由于不需要去创建 `bean`，因此可以直接从 `AbstractApplicationContext` 的 `beans` 属性中得到：

```
public Object getBeanIgnoreCreate(String id) {
    return this.beans.get(id);
}
```

现在，IoC 容器创建 `bean` 的方法已经全部实现。

#### 14.7.4 为 AbstractApplicationContext 添加子类

`AbstractApplicationContext` 是一个抽象类，并不可以实例化，因此我们为它添加一个子类 `XmlApplicationContext`。

代码清单：code\IoC\main\org\crazyit\ioc\context\XmlApplicationContext.java

```
public class XmlApplicationContext extends AbstractApplicationContext {
    public XmlApplicationContext(String[] xmlPaths) {
        //初始化文档和元素
        setUpElements(xmlPaths);
    }
}
```

我们为 `XmlApplicationContext` 添加一个构造器，参数是多份 `xml` 文件的路径，得到参数后，再调用父类的 `setUpElements` 将 `Document` 和 `Element` 缓存。我们在父类的实现中，只是将多个 `Document` 和 `Element` 对象缓存，并没有创建任何的 `bean`，因此，我们需要为 `AbstractApplicationContext` 添加一个创建 `bean` 的方法，当加载完配置文件后，就马上启动 IoC 容器创建 `bean`。

代码清单：code\IoC\main\org\crazyit\ioc\context\AbstractApplicationContext.java

```
//创建所有的 bean 实例，延迟加载的不创建
protected void createBeans() {
    Collection<Element> elements = elementLoader.getElements();
    for (Element e : elements) {
        boolean lazy = elementReader.isLazy(e);
        //如果不是延迟加载，再判断是否单态
        if (!lazy) {
            String id = e.attributeValue("id");
            Object bean = this.getBean(id);
            if (bean == null) {
                //处单 bean，如果是单态的，加到缓存中，非单态则不创建
                handleSingleton(id);
            }
        }
    }
}
```

为 `AbstractApplicationContext` 添加了一个 `createBeans` 方法，用于在容器初始化的时候创建 `bean`。注意以上代码的黑体部分，调用了 `AbstractApplicationContext` 的私有方法 `handleSingleton` 方法创建 `bean`。

那么我们可以在 `XmlApplicationContext` 的构造器中加入 `createBeans` 方法：

```
public XmlApplicationContext(String[] xmlPaths) {
```

```
//初始化文档和元素
setUpElements(xmlPaths);
//初始化容器
createBeans();
}
```

这样，当 `XmlApplicationContext` 进行构造时，就会初始化文档和元素，并初始化 IoC 容器。现在再为 `AbstractApplicationContext` 新建一个子类 `XmlBeanFactory`，该类的作用主要是当构造的时候，只负责读取 XML 配置文件，不初始化容器，如果调用 `getBean` 方法时，才创建 bean。

代码清单：code\IoC\main\org\crazyit\ioc\beans\factory\XmlBeanFactory.java

```
public class XmlBeanFactory extends AbstractApplicationContext {
    public XmlBeanFactory(String[] xmlPaths) {
        //只初始化文档，不创建任何 bean
        setUpElements(xmlPaths);
    }
}
```

### 14.7.5 测试IoC容器的创建

在上一小节中，我们已经确定了 `AbstractApplicationContext` 有两个子类，那么就代表我们的 IoC 容器有两种创建方式，可以在初始化容器的时候，就创建所有的 bean，也可以不创建任何的 bean，让外界调用 `getBean` 方法的时候再创建。到了现在，整个 IoC 容器的功能都已经实现了，我们可以编写相关的代码对这些功能进行测试。

首先简单的测试 `getBean` 方法，以下是测试的准备数据：

```
<bean id="test1" class="org.crazyit.ioc.context.object.XmlApplicationContextObject1"></bean>
```

以上的配置定义了一个 test1 的 bean，类 `XmlApplicationContextObject` 只是一个普通的 Java 类。下面的代码使用 `XmlApplicationContext` 来创建 IoC 容器：

```
ApplicationContext ctx = new XmlApplicationContext
(new String[]{"resources/context/XmlApplicationContext1.xml"});
```

现在可以调用 `getBean` 方法获得 test1 的 bean，将得到的对象打印出来查看是否有值，以下为测试的代码：

```
//拿到 test1，使用无参构造器创建
XmlApplicationContextObject1 obj1 = (XmlApplicationContextObject1)ctx.getBean("test1");
System.out.println(obj1);
```

输出结果：

```
org.crazyit.ioc.context.object.XmlApplicationContextObject1 @56a499
```

我们的 `getBean` 方法可以帮我们创建 bean 了，但是，我们提供的配置文件中，只有简单的 id 和 class 属性来定义 bean，这里没有配置任何的构造参数，并且没有配置任何属性，因此我们的程序是通过调用 bean 的无参数构造器来创建实例。下面我们测试 bean 的 singleton 属性（是否单态）。

以下为测试的配置：

```
<!-- 这是单态的 bean(默认为单态) -->
<bean id="test1" class="org.crazyit.ioc.context.object.XmlApplicationContextObject1"></bean>
<!-- 这是非单态的 bean -->
<bean id="test3" class="org.crazyit.ioc.context.object.XmlApplicationContextObject1"
    singleton="false"></bean>
```

测试代码：

```
//test1 是单态 bean
XmlApplicationContextObject1 obj1 = (XmlApplicationContextObject1)ctx.getBean("test1");
XmlApplicationContextObject1 obj2 = (XmlApplicationContextObject1)ctx.getBean("test1");
```

```

System.out.println(obj1);
System.out.println(obj2);
//test3 不是单态 bean
XmlApplicationContextObject1 obj3 = (XmlApplicationContextObject1)ctx.getBean("test3");
XmlApplicationContextObject1 obj4 = (XmlApplicationContextObject1)ctx.getBean("test3");
System.out.println(obj3);
System.out.println(obj4);

```

打印结果:

```

org.crazyit.ioc.context.object.XmlApplicationContextObject1 @1e51060
org.crazyit.ioc.context.object.XmlApplicationContextObject1 @1e51060
org.crazyit.ioc.context.object.XmlApplicationContextObject1 @a1807c
org.crazyit.ioc.context.object.XmlApplicationContextObject1 @fa7e74

```

从结果中可以看到 **test3** 的 bean 是非单态的, 因此打印结果的最后两行, 两个对象的 hashCode 并不一致, 下面测试构造注入。

以下为测试的配置:

```

<!-- test1 没有构造参数 -->
<bean id="test1" class="org.crazyit.ioc.context.object.XmlApplicationContextObject1"></bean>
<!-- test2 使用有参数的构造器, 其中一个构造参数是 test1 -->
<bean id="test2" class="org.crazyit.ioc.context.object.XmlApplicationContextObject2">
    <constructor-arg>
        <value type="java.lang.String">yangenxiong</value>
    </constructor-arg>
    <constructor-arg>
        <value type="java.lang.Integer">10</value>
    </constructor-arg>
    <constructor-arg>
        <ref bean="test1"/>
    </constructor-arg>
</bean>

```

注意以上配置的黑体部分, **test2** 其中的一个构造参数使用的是 **test1** 的 bean, 以下为测试代码:

```

XmlApplicationContextObject1 obj1 = (XmlApplicationContextObject1)ctx.getBean("test1");
//拿到第二个 bean, 使用多参数构造器创建
XmlApplicationContextObject2 obj2 = (XmlApplicationContextObject2)ctx.getBean("test2");
//打印 test2 的 name
System.out.println(obj2.getName());
//打印 test2 的 age
System.out.println(obj2.getAge());
//打印 test1 的 bean
System.out.println(obj1);
//打印 test2 的 object1 属性
System.out.println(obj2.getObject1());

```

可以看到效果如下:

```

yangenxiong
10
org.crazyit.ioc.context.object.XmlApplicationContextObject1 @183f74d
org.crazyit.ioc.context.object.XmlApplicationContextObject1 @183f74d

```

可以看到结果的最后两行, 我们从容器中直接通过 **getBean** 方法得到 **test1** 的实例, 再通过 **getBean** 方法得到 **test2** 的实例, 这里再通过 **test2** 获得它的 **object1** 属性, 可以看到该属性的值与 **test1** 的实例一样。

注: 这里的 XmlApplicationContextObject1 对象和 XmlApplicationContextObject2 对象都是普通的 Java 对象, XmlApplicationContextObject2 中有一个属性叫 object1, 是 XmlApplicationContextObject1 类型。

下面继续测试自动装配, 以下为测试配置:

```
<!-- object1 不自动装配 -->
<bean class="org.crazyit.ioc.context.object.XmlApplicationContextObject1"
      id="object1">
</bean>
<!-- test4 使用名字自动装配 -->
<bean id="test4"
      class="org.crazyit.ioc.context.object.XmlApplicationContextObject3"
      autowire="byName">
</bean>
```

以上的配置中, object1 不自动装配, test4 使用自动装配, XmlApplicationContextObject1 对象里面没有任何的属性, XmlApplicationContextObject3 的属性如下:

```
public class XmlApplicationContextObject3 {
    private String name;
    private int age;
    private XmlApplicationContextObject1 object1;
}
```

我们配置了 test4 为自动装配, 因此得到 test4 的实现时, 就可以打印 XmlApplicationContextObject3 的 object1 看下是否有值, 以下为测试代码:

```
XmlApplicationContextObject3 obj1 = (XmlApplicationContextObject3)ctx.getBean("test4");
System.out.println(obj1);
XmlApplicationContextObject1 obj2 = obj1.getObject1();
System.out.println(obj2);
XmlApplicationContextObject1 obj3 = (XmlApplicationContextObject1)ctx.getBean("object1");
System.out.println(obj3);
```

以上的测试代码中, 先从容器中得到一个 XmlApplicationContextObject3 对象, 再通过该对象的 getObject1 方法得到 XmlApplicationContextObject1 对象, 再从容器中通过 getBean 方法得到 XmlApplicationContextObject1 对象, 以下为打印结果, 可以看到以上的黑体的两个实例的 hashCode 相同:

```
org.crazyit.ioc.context.object.XmlApplicationContextObject3@124bbbf
org.crazyit.ioc.context.object.XmlApplicationContextObject1@1fdc96c
org.crazyit.ioc.context.object.XmlApplicationContextObject1@1fdc96c
```

从以上的结果可以看出, test4 的 bean 没有配置任何的属性, 就可以通过名字进行自动装配, 将容器中的 bean 设置到实例里面。

下面测试不通过自动装配进行设值注入, 以下为测试的配置:

```
<!-- object1 的 bean -->
<bean class="org.crazyit.ioc.context.object.XmlApplicationContextObject1"
      id="object1">
</bean>
<!-- test6 的 bean, 有一个属性为 object1 -->
<bean id="test6" class="org.crazyit.ioc.context.object.XmlApplicationContextObject3">
    <property name="name">
        <value type="java.lang.String">yangexiong</value>
    </property>
    <property name="age">
        <value type="java.lang.Integer">10</value>
    </property>
```

```

        <property name="object1">
            <ref bean="object1"/>
        </property>
    </bean>

```

注意：我们在本节中所使用的 XmlApplicationContextObject1、XmlApplicationContextObject2 与 XmlApplicationContextObject3 对象都是同一个。

我们在前面已经测试了根据名字自动装配，我们在这里测试不使用自动装配进行设值注入，与前面的测试代码区别不大：

```

XmlApplicationContextObject3 obj1 = (XmlApplicationContextObject3)ctx.getBean("test6");
XmlApplicationContextObject1 obj2 = (XmlApplicationContextObject1)ctx.getBean("object1");
System.out.println(obj1.getName());
System.out.println(obj1.getAge());
System.out.println(obj1.getObject1());
System.out.println(obj2);

```

以下为打印结果：

```

yangenxiong
10
org.crazyit.ioc.context.object.XmlApplicationContextObject1@13bad12
org.crazyit.ioc.context.object.XmlApplicationContextObject1@13bad12

```

通过 XmlApplicationContextObject3 的实例得到 XmlApplicationContextObject1 的实例时，可以看到得到的 hashCode 与直接到容器中 getBean 得到的 hashCode 一致。

测试延迟加载，以下为没有的配置：

```

<bean id="test5" class="org.crazyit.ioc.context.object.XmlApplicationContextObject3" lazy-init="true">
</bean>

```

以上的配置只有一个 bean，该 bean 是需要延迟加载的，我们可以调用 ApplicationContext 的 getBeanIgnoreCreate 方法得到 bean，因此该方法会从容器中查找 bean，如果不能找到，则返回 null，并不会帮我们去创建，以下为测试代码：

```

//test5 是延迟加载的，没有调用过 getBean 方法，那么容器中就不会创建这个 bean
Object obj = ctx.getBeanIgnoreCreate("test5");
System.out.println(obj);
obj = ctx.getBean("test5");
System.out.println(obj);

```

在创建容器的时候 test5 并不会被创建，因此它是延迟加载的，当我们再次调用 getBean 方法得到 test5 时，可以发现 getBean 方法实际上已经帮我们创建了实例，以下为打印结果：

```

null
org.crazyit.ioc.context.object.XmlApplicationContextObject3@1c39a2d

```

下面我们可以测试 XmlBeanFactory 类，如果使用该类初始化容器，那么所有的 bean 都不会被创建，以下为测试配置：

```

<beans>
    <bean id="test1" class="org.crazyit.ioc.context.object.XmlApplicationContextObject1"></bean>
</beans>

```

以上定义了一个名字叫 test1 的 bean，如果使用 XmlApplicationContext 来创建容器的话，那么这个 bean 在容器启动的时候就会被创建，但是如果使用 XmlBeanFactory 创建容器的话，该 bean 不会被创建，以下为测试代码：

```

ApplicationContext ctx = new XmlBeanFactory(new String[]{"resources/factory/XmlBeanFactory.xml"});
//容器初始化的时候不会创建
Object obj = ctx.getBeanIgnoreCreate("test1");
System.out.println(obj);
//调用 getBean 方法，才会创建

```



```
obj = ctx.getBean("test1");  
System.out.println(obj);  
可以看到打印结果如下:
```

```
null
```

```
org.crazyit.ioc.context.object.XmlApplicationContextObject1@1754ad2
```

到此，我们 IoC 的相关功能都已经实例了，并且都经过了测试，那么这个 IoC 可以给我们带来什么方便呢，在下一节，我们使用在这一节中编写的 IoC[容器与第 9 章图书进销系统进行整合，来体会我们编写的这个 IoC 所带来的好处。

## 14.8 IoC与图书进销系统的整合

我们在前面编写了一个简单的 IoC 容器，我们可以配置各个对象，并为它们配置属性或者构造参数，我们可以将各个对象作为容器中的一个 bean，让 IoC 容器帮我们去创建它们的实例和对它们进行管理，我们可以不需要使用 new 关键字来创建类的实例，这一节，我们来体会这个 IoC 容器带我们带来的好处。在阅读本节前，可以先去了解第 9 章图书进销系统的分层结构。

### 14.8.1 需要管理的对象

在整合前，我们需要明白哪些对象需要给 IoC 容器进行管理的。第 9 章图书进销系统中，我们对程序进行分层，有视图层、业务处理层和数据访问层。视图层包括各个 JPanel 对象和两个 JFrame 对象，业务处理层（Service）有各个业务处理的接口和对应的实现类，数据访问层（DAO）有各类数据的访问接口和实现。

视图层与业务层的关系，请看图 14.1。

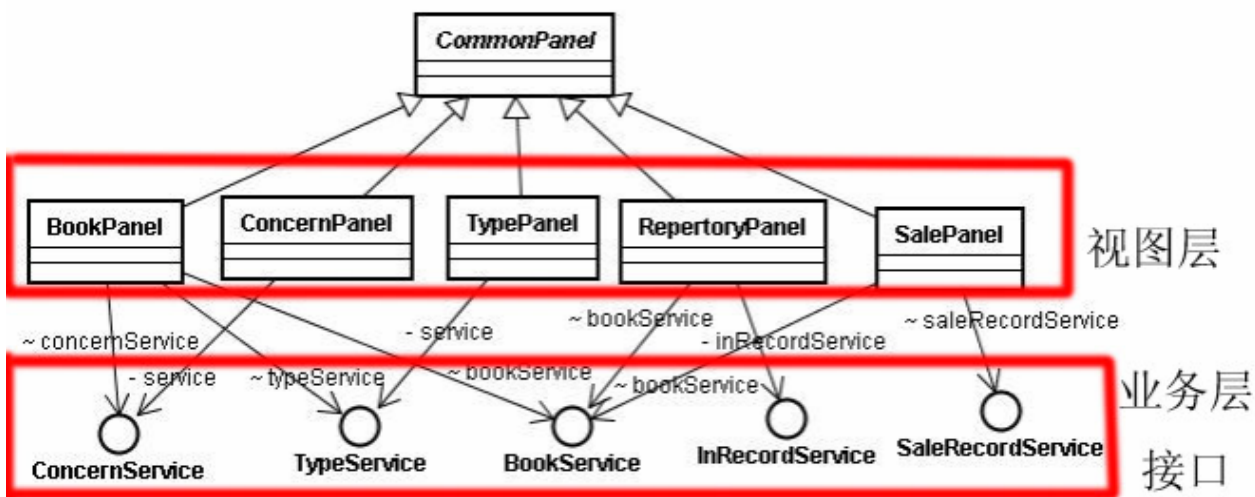


图 14.1 视图层与业务层类图

从图 14.1 中我们可以看到，图书进销系统中，各个视图对象里面都有一个或者多个业务层接口对象，那么如果我们使用 IoC 来创建业务层对象的话，那么在视图就可以不必手动去创建这些对象。另外，各个视图层的 JPanel 对象，都是给主界面 MainFrame 来使用的，因此也可以将这些视图层的 JPanel 对象做成 bean，并注入到 MainFrame 中。

业务层的实现与数据访问层的关系，如图 14.2 所示。

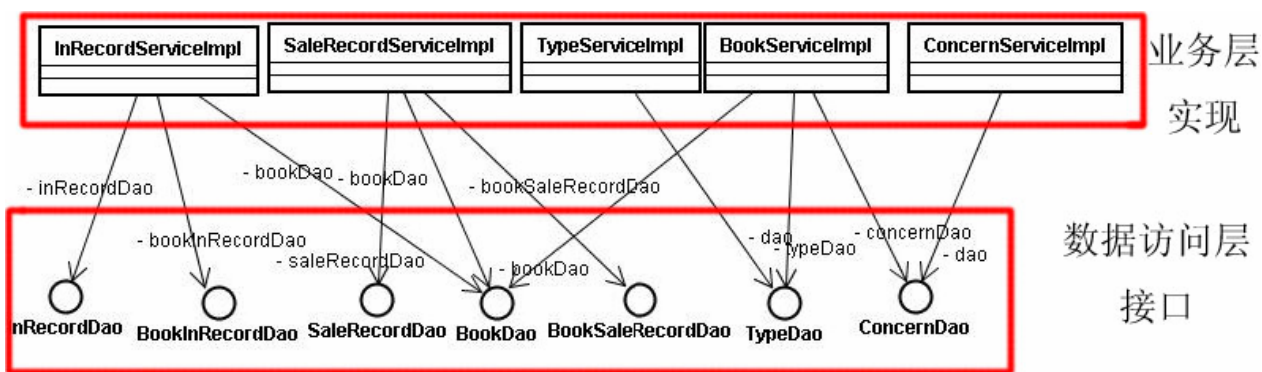


图 14.2 业务层实现与数据访问层的类图

从图 14.2 可以看到，业务层的各个实现类，里面都使用到一个或者多个的数据访问层接口，因此，这数据访问层我们都可以交给 IoC 容器去管理。

### 14.8.2 创建数据访问层的bean

首先，我们需要将本章所编写的 IoC 容器打成一个 jar 包，然后放到图书进销系统的 classpath 中，在项目的 src 目录下新建 resources 目录，新建 daoContext.xml 配置文件，daoContext.xml 文件的内容如下。

代码清单：code\book(IoC)\src\resource\daoContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//CRAZYIT/DTD BEAN/EN" "http://www.crazyit.org/beans.dtd">
<beans>
    <!-- DAO -->
    <!-- 用户的 dao -->
    <bean id="userDao" class="org.crazyit.book.dao.impl.UserDaoImpl"></bean>
    <!-- 用户的业务接口，使用设值注入 -->
    <!-- 书本种类 DAO -->
    <bean id="typeDao" class="org.crazyit.book.dao.impl.TypeDaoImpl"></bean>
    <!-- 出版社 DAO -->
    <bean id="concernDao" class="org.crazyit.book.dao.impl.ConcernDaoImpl"></bean>
    <!-- 书本 DAO -->
    <bean id="bookDao" class="org.crazyit.book.dao.impl.BookDaoImpl"></bean>
    <!-- 销售 DAO -->
    <bean id="saleRecordDao" class="org.crazyit.book.dao.impl.SaleRecordDaoImpl"></bean>
    <!-- 书本销售 DAO -->
    <bean id="bookSaleRecordDao" class="org.crazyit.book.dao.impl.BookSaleRecordDaoImpl"></bean>
    <!-- 入库 DAO -->
    <bean id="inRecordDao" class="org.crazyit.book.dao.impl.InRecordDaoImpl"></bean>
    <!-- 书本入库 DAO -->
    <bean id="bookInRecordDao" class="org.crazyit.book.dao.impl.BookInRecordDaoImpl"></bean>
</beans>
```

由于我们的 IoC 容器支持多份配置文件，因此可以将数据访问层、业务层和视图层分开配置，这样也比较清晰，以上的配置中，直接定义各个 DAO 的 bean，指定 id 和 class 即可。

### 14.8.3 创建业务层的bean

接下来定义业务层的配置，业务层需要注入各个 DAO 的 bean，业务层接口的实现类使用了一个或者多个的数据访问层接口，因此我们在配置业务层的 bean 的时候，可以使用根据名字自动装配 bean（`autowire="byname"`），具体配置如下。

代码清单：code\book(IoC)\src\resource\serviceContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//CRAZYIT//DTD BEAN//EN" "http://www.crazyit.org/beans.dtd">
<beans>
    <!-- Service，在这里使用自动装配 -->
    <!-- 用户 service 使用设值注入，不自动装配 -->
    <bean id="userService" class="org.crazyit.book.service.impl.UserServiceImpl">
        <property name="userDao">
            <ref bean="userDao"/>
        </property>
    </bean>
    <!-- 种类 service -->
    <bean id="typeService"
        class="org.crazyit.book.service.impl.TypeServiceImpl" autowire="byName"></bean>
    <!-- 出版社 service -->
    <bean id="concernService"
        class="org.crazyit.book.service.impl.ConcernServiceImpl" autowire="byName"></bean>
    <!-- 书本 service -->
    <bean id="bookService"
        class="org.crazyit.book.service.impl.BookServiceImpl" autowire="byName"></bean>
    <!-- 销售 service -->
    <bean id="saleRecordService"
        class="org.crazyit.book.service.impl.SaleRecordServiceImpl" autowire="byName"></bean>
    <!-- 入库 service -->
    <bean id="inRecordService"
        class="org.crazyit.book.service.impl.InRecordServiceImpl" autowire="byName"></bean>
</beans>
```

注意，为了展示我们的设值注入功能，上面的业务层的各个 bean 配置中，除了 userService 没有使用自动装配外，其他的 bean 都使用了自动装配。

为业务层的各个组件定义了配置后，我们还需要修改实现类的代码（`ServiceImpl`），原来是使用构造器的方式进行设值，现在可以不使用构造器设值，只需要为各个 DAO 的属性提供 setter 方法，setter 方法的名字必须是 set 打头，后面跟 DAO bean 的 id。

原来的业务层代码：

```
private BookDao bookDao;
private TypeDao typeDao;
private ConcernDao concernDao;
public BookServiceImpl(BookDao bookDao, TypeDao typeDao,
    ConcernDao concernDao) {
    this.bookDao = bookDao;
    this.typeDao = typeDao;
    this.concernDao = concernDao;
}
```

修改后的业务层代码：

```
private BookDao bookDao;
```

```

private TypeDao typeDao;
private ConcernDao concernDao;
public void setBookDao(BookDao bookDao) {
    this.bookDao = bookDao;
}
public void setTypeDao(TypeDao typeDao) {
    this.typeDao = typeDao;
}
public void setConcernDao(ConcernDao concernDao) {
    this.concernDao = concernDao;
}
}

```

注意 setter 方法的命名，set (DAO bean name)。

#### 14.8.4 创建视图层的bean

接下来创建视图层的配置文件 `uiContext.xml`，在这里需要注意，各个视图层在构造的时候，就需要得到某个业务层对象，调用业务层的对象进行初始化数据，因此需要使用构造注入，以下为具体配置。

代码清单：`code\book\IoC\src\resource\uiContext.xml`

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//CRAZYIT//DTD BEAN//EN" "http://www.crazyit.org/beans.dtd">
<beans>
    <!-- 界面对象 -->
    <!-- 登录界面，同时使用构造注入和设置注入 -->
    <bean id="loginFrame" class="org.crazyit.book.ui.LoginFrame">
        <!-- 通过构造注入，向 LoginFrame 注入 userService(用户业务接口) -->
        <constructor-arg>
            <ref bean="userService"/>
        </constructor-arg>
        <!-- 通过设置注入方式，向 LoginFrame 对象注入 MainFrame 的 bean -->
        <property name="mainFrame">
            <ref bean="mainFrame"/>
        </property>
    </bean>
    <!--
        主界面 MainFrame 的 bean，需要进行构造注入，
        由于初始化界面的时候，需要设置当前界面为销售界面，
        除了 SalePanel 外，其他界面对象通过自动装配初始化
    -->
    <bean id="mainFrame" class="org.crazyit.book.ui.MainFrame" autowire="byName">
        <constructor-arg>
            <ref bean="salePanel"/>
        </constructor-arg>
    </bean>
    <!-- 各个界面对象(JPanel 对象)，在这里使用构造注入，因为在创建这些对象时，
        需要初始化数据，数据是从业务层得到的 -->
    <!-- 销售 JPanel -->
    <bean id="salePanel" class="org.crazyit.book.ui.SalePanel">
        <constructor-arg>
            <ref bean="bookService"/>

```

```

        </constructor-arg>
        <constructor-arg>
            <ref bean="saleRecordService"/>
        </constructor-arg>
    </bean>
    <!-- 库存 JPanel -->
    <bean id="repertoryPanel" class="org.crazyit.book.ui.RepertoryPanel">
        <constructor-arg>
            <ref bean="inRecordService"/>
        </constructor-arg>
        <constructor-arg>
            <ref bean="bookService"/>
        </constructor-arg>
    </bean>
    <!-- 书本 JPanel -->
    <bean id="bookPanel" class="org.crazyit.book.ui.BookPanel">
        <constructor-arg>
            <ref bean="bookService"/>
        </constructor-arg>
        <constructor-arg>
            <ref bean="typeService"/>
        </constructor-arg>
        <constructor-arg>
            <ref bean="concernService"/>
        </constructor-arg>
    </bean>
    <!-- 出版社 JPanel -->
    <bean id="concernPanel" class="org.crazyit.book.ui.ConcernPanel">
        <constructor-arg>
            <ref bean="concernService"/>
        </constructor-arg>
    </bean>
    <!-- 种类 JPanel -->
    <bean id="typePanel" class="org.crazyit.book.ui.TypePanel">
        <constructor-arg>
            <ref bean="typeService"/>
        </constructor-arg>
    </bean>
</beans>

```

注意以上配置中 `mainFrame` 的 `bean`，由于 `MainFrame` 在创建的时候，就需要马上显示销售的界面，因此我们将销售界面的 `bean` 构造注入到 `MainFrame` 中。现在可以修改 `MainFrame` 中各个对象的创建，原来直接使用 `new` 关键字在 `MainFrame` 中创建各个组件。

代码清单：code\book\IoC\src\org\crazyit\book\ui\MainFrame

```

public MainFrame() {
    //创建 DAO 对象
    TypeDao typeDao = new TypeDaoImpl();
    ConcernDao concernDao = new ConcernDaoImpl();
    BookDao bookDao = new BookDaoImpl();
    SaleRecordDao saleRecordDao = new SaleRecordDaoImpl();
    BookSaleRecordDao bookSaleRecordDao = new BookSaleRecordDaoImpl();
}

```

```

InRecordDao inRecordDao = new InRecordDaoImpl();
BookInRecordDao bookInRecordDao = new BookInRecordDaoImpl();
//创建 Service
this.typeService = new TypeServiceImpl(typeDao);
this.concernService = new ConcernServiceImpl(concernDao);
this.bookService = new BookServiceImpl(bookDao, typeDao, concernDao);
this.saleRecordService = new SaleRecordServiceImpl(saleRecordDao,
    bookSaleRecordDao, bookDao);
this.inRecordService = new InRecordServiceImpl(inRecordDao, bookInRecordDao, bookDao);
//让销售界面作为第一显示界面
this.salePanel = new SalePanel(this.bookService, this.saleRecordService);
//初始化库存管理界面
repertoryPanel = new RepertoryPanel(this.inRecordService, this.bookService);
//初始化书本管理界面
bookPanel = new BookPanel(this.bookService, this.typeService,
    this.concernService);
//初始化出版社管理界面
concernPanel = new ConcernPanel(this.concernService);
//初始化种类管理界面
typePanel = new TypePanel(this.typeService);
//省略其他代码
}

```

那么现在可以修改成：

```

public MainFrame(SalePanel salePanel) {
    this.salePanel = salePanel;
    //省略其他代码
}

```

修改了 `MainFrame` 的构造器后，还需要为其他的视图层对象（`JPanel`）添加 `setter` 方法，`setter` 方法的命名与业务层实现类中的 `setter` 方法命名一样，`set`（视图层 `bean` 的 `id`）。再以同样的方法去修改 `LoginFrame`，为 `LoginFrame` 构造注入一个 `UserService` 的 `bean`，设值注入一个 `MainFrame` 的 `bean`，下面是 `LoginFrame` 的 `login` 方法的部分代码：

```

try {
    userService.login(name, password.toString());
    //成功登录，显示 MainFrame
    this.mainFrame.setVisible(true);
    //隐藏 LoginFrame
    this.setVisible(false);
} catch (Exception e) {
    e.printStackTrace();
    showWarn(e.getMessage());
}

```

由于我们在 `LoginFrame` 的 `login` 方法中设定了，如果登录成功的话，才显示 `MainFrame`，那么就需要修改 `MainFrame` 的构造器，删除 `this.setVisible(true)` 一句。最后，我们在 `Main` 类（程序启动类）中加入初始化 `IoC` 容器的代码：

```

public static void main(String[] args) {
    //创建容器
    ApplicationContext ctx = new XmlApplicationContext(new String[]{"resource/daoContext.xml",
        "resource/serviceContext.xml", "resource/uiContext.xml"});
    //得到 LoginFrame 的实例
    LoginFrame loginFrame = (LoginFrame)ctx.getBean("loginFrame");
}

```

```
//设置 LoginFrame 可见  
loginFrame.setVisible(true);  
}
```

到现在，与图书进销系统的整合已经完成，在整合的过程中，我们可以看到，以前所编写的一些创建实例的代码可以全部省略，我们可以不再关心各个实例的创建，这些创建实例的行为，全部交由给 IoC 容器去实现，让容器为我们去管理各个系统中的组件，将代码的耦合降到了配置文件中。

## 14.9 本章小结

在本章我们使用了 dom4j、Java 的反射机制编写了一个简单的 IoC 容器，最后并与现有系统进行整合，展示 IoC 容器给我们带来的价值。一个强大的 IoC 容器还包括 bean 的继承、多种自动装配方式（本章只有 byName 自动装配）等等一系列功能，相对于当前流行的一些 IoC 容器，本章所实现的基本功能是远远不够的，但是通过本章所介绍的一些 IoC 最基本的功能，希望可以给读者带来启发，让大家领会到 IoC 的工作原理，在使用其他流行的 IoC 容器的时候，除了会使用外，还可以懂得原理，加深对 IoC 的认识。

## 第 15 章 仿QQ游戏大厅

### 15.1 游戏大厅简介

我们曾经见过许多的游戏大厅，笔者从最开始接触的联众游戏大厅到现在十分多人玩的 QQ 游戏大厅，这些游戏大厅为我们提供了游戏的平台，可以让我们在网络上进行各种游戏对战。这些游戏大厅提供了各式各样的游戏，例如斗地主、泡泡龙、俄罗斯方块等，我们只要下载一些游戏大厅的客户端，就可以进行网络的游戏对战，并从游戏中得到积分。在玩这些游戏大厅的时候，我们不妨可以考虑一下，使用我们所学的 Java 知识去实现这些游戏大厅，本章主要介绍如何使用 Java 去开发一款属于自己的游戏大厅，让大家在开发的过程中，了解这些游戏大厅的实现原理。

### 15.2 编写游戏大厅框架

在开发游戏大厅前，我们需要先了解下游戏大厅的原理。例如，一个玩家登录进入游戏大厅，那么就需要将该用户进入的信息发送给其他已经进入大厅的玩家，如果玩家坐下到大厅的某个桌子的时候，就需要将这些信息（玩家坐下的信息）告诉其他玩家，并更新其他玩家的界面组件。在本章中我们使用 Socket 来进行服务器端与客户端之间的通信，Socket 就是两台机器上的通信端点。

Socket 中包含一个输出流对象，一台机器可以通过这个输出流，将信息发送到另外一端的机器中。例如，我们可以使用以下代码得到输出流，并将一些信息打印到输出流中：

```
PrintStream ps = new PrintStream(socket.getOutputStream());  
ps.println("这是打印信息");
```

使用以上的方法，可以将一些信息从一台机器发送到另外一台机器中。我们在游戏大厅中的每一个操作，都可能会将自己所操作的信息发送到服务器，再由服务器转发给其他玩家，因此在玩家发送信息给服务器或者服务器再转发给其他玩家，都可以使用 PrintStream 的 print 方法来将信息打印到 Socket 的输出流中。既然每个操作都如此，我们可以编写一个游戏大厅的小框架，专门用于中转，框架中的服务器端与客户端的代码都不需要编写任何的逻辑，如果编写了新的游戏，可以将该游戏的包放到框架中运行，当然，这些游戏都必须遵守框架的规则。

#### 15.2.1 确定传输格式

我们已经知道了服务器间通过 PrintStream 来向输出流打印字符串信息，那么我们可以先确定这些字符串的格式。本章中使用 XML 作为它们之间的传输格式，例如服务器向客户端输出信息的时候，可以将一段 XML 打印到输出流中，客户端得到这些 XML 字段串后，就将其转化为特定的对象，再根据这些对象进行相应的处理。

当客户端发送一些信息给服务器的时候，我们就把这些信息封装到一个 Request 对象中，该对象包括参数列表、服务器处理类、客户端处理类等信息，当服务器端接收到该请求对象后，就将这些信息返回一个 Response 对象，将 Request 与 Response 对象当作参数传递给服务器处理类，当服务器处理类完成了服务器的操作后，就将这些信息通过一个 Response 对象返回给客户端。

代码清单：code\GameHall-Commons\src\org\crazyit\gamehall\commons\Request.java

```
public class Request {  
    //参数列表
```



```

private Map<String, Object> parameters;
//服务器处理类
private String serverActionClass;
//客户端处理类
private String clientActionClass;
}

```

代码清单: code\GameHall-Commons\src\org\crazyit\gamehall\commons\Response.java

```

public class Response {
    //服务器返回的各个数值
    private Map<String, Object> datas;
    //错误代码
    private String errorCode;
    //客户端处理类, 该值保存在 Request 的请求参数的 Map 中
    private String actionClass;
}

```

当客户端发送一次请求的时候, 就封装一个 **Request** 对象, 告诉服务器由哪个服务器处理类处理一次的请求。**Request** 对象中的服务器处理类与客户端处理类由发送请求的客户来确定, 因此客户端需要清楚的知道服务器处理类的具体类名。

这样就好比我们在开发 **web** 应用的时候, 客户端发送一个请求的 **url**, 服务器就可以根据这个 **url** 来确定处理类。确定了请求的对象为 **Request**, 服务器响应的对象为 **Response** 后, 客户端发送请求的字段串时 (在将 **XML** 打印到 **Socket** 输出流), 就可以将 **Request** 对象转化为一个 **XML** 字符串, 服务器作为响应时, 就可以将一个 **Response** 对象转化为一个 **XML** 字符串, 服务器或者客户端得到该 **XML** 字符串后, 就可以转化为 **Request** 或者 **Response** 对象。对象与 **XML** 之间的转换我们使用 **XStream** 来实现, **XStream** 可以轻松的帮我们在这两者之间进行转换。

编写一个 **XStreamUtil** 的类, 用于处理对象与 **XML** 之间的转换。

代码清单: code\GameHall-Commons\src\org\crazyit\gamehall\util\XStreamUtil.java

```

public class XStreamUtil {
    private static XStream xstream = new XStream();
    //将 XML 转换成对象
    public static Object fromXML(String xml) {
        return xstream.fromXML(xml);
    }
    //将对象转换成 XML 字段串
    public static String toXML(Object obj) {
        String xml = xstream.toXML(obj);
        //去掉换行
        String a = xml.replaceAll("\n", "");
        String s = a.replaceAll("\r", "");
        return s;
    }
}

```

需要注意的是, 将对象转换成 **XML** 的时候, 需要将生成的 **XML** 字符串进行处理, 去掉换行。

## 15.2.2 建立处理类接口

当服务器接收到客户端发送的一次请求后, 就可以根据 **Request** 对象得到服务器处理类, 我们先编写服务器处理类的接口。

新建 **ServerAction** 接口。

代码清单: code\GameHall-Commons\src\org\crazyit\gamehall\commons\ServerAction.java

```
//服务器处理请求的接口
public interface ServerAction {
    //Action 的执行方法
    void execute(Request request, Response response, Socket socket);
}
```

通过 **Request** 具体的某个服务器处理类后，就可以使用 **Java** 的反射来实现化该类，由于服务器处理类都必须实现 **ServerAction** 接口，因此实现化该类后，就可以直接调用 **execute** 方法来执行某些具体的行为。

同样的，新建一个 **ClientClass** 接口，表示客户端处理类。**ClientAction**。

代码清单：code\GameHall-Commons\src\org\crazyit\gamehall\commons\ClientAction.java

```
//客户端处理服务器响应的接口
public interface ClientAction {
    //客户端处理类执行
    void execute(Response response);
}
```

服务器端发送响应到客户端时，客户端就根据响应（**Response**）中的处理类来执行某些操作，例如更新界面组件等。**Response** 对象中包含一个 **actionClass** 的处理类，该类就是客户端的处理类。得到这个处理类的字符串后，同样地，我们就可以根据这个类名来得到具体的某个 **ClientClass** 的实现类实例，再调用 **execute** 方法即可。

## 15.2.2 建立玩家类与游戏接口

我们为游戏大厅新建一个玩家类，用来代表一个玩家，该类中保存了玩家的大多数信息，包括名称、头像图片、该玩家的标识等。

代码清单：code\GameHall-Commons\src\org\crazyit\gamehall\commons\User.java

```
public class User {
    //玩家的唯一标识
    private String id;
    //头像图片
    private String headImage;
    //玩家名称
    private String name;
    //0 男, 1 女
    private int sex;
    //玩家对应的 Socket
    private Socket socket;
}
```

**User** 类中除了包含玩家的一些基本信息外，还需要保存一个 **Socket** 对象，在服务器中我们需要将这一系列的玩家对象都保存起来，当根据玩家的 **id** 得到某个玩家对象后，就可以直接得到该玩家对应的 **Socket**，根据这个 **Socket** 就可以向客户端发送相关的信息（**Response** 对象）。在实际的情况中，每个游戏都会有不同的玩家对象，因此当我们编写一个新的游戏并放到这个框架中的时候，可以根据游戏的实际情况来继承 **User** 类。

接下来新建一个游戏的接口，该接口只需要提供一个 **start** 的方法，表示游戏的开始动作，因此，每一个放在该框架中的游戏，都需要去实现这个游戏接口，并实现 **start** 方法。

代码清单：code\GameHall-Commons\src\org\crazyit\gamehall\commons\Game.java

```
public interface Game {
    //开始游戏的方法
    void start(User user);
}
```

```
}
```

到这里，我们的这个框架定义了两个规范，第一，客户端与服务器进行信息传输的时候，只能使用 **Request** 和 **Response** 对象所生成的 XML 字符串；第二，每个放在该框架中的游戏都必须提供一个实现 **Game** 接口的游戏实现类，为游戏提供一个入口。

### 15.2.3 编写框架服务器

服务器端编写较为简单，只需要创建一个 **ServerSocket** 对象，再使用该对象的 **accept** 方法来监听和接收连接到服务器的 **Socket**，再以该 **Socket** 来启动一条线程来监听客户端所发送的请求。

代码清单：code\GameHall-Server\src\org\crazyit\gamehall\server\Server.java

```
public class Server {
    //服务器 Socket 对象
    ServerSocket serverSocket;
    public Server() {
        //创建 ServerSocket 对象, 端口为 12000
        this.serverSocket = new ServerSocket(12000);
        while(true) {
            //监听 Socket 的连接
            Socket s = this.serverSocket.accept();
            //启动服务器线程
            new ServerThread(s).start();
        }
    }
}
```

该类接收到 **Socket** 后，就启动服务器线程，用于处理服务器端 **Socket** 接收到的信息。那么服务器线程就可以一直监听建立这个 **Socket** 的客户端所发送的信息。

代码清单：code\GameHall-Server\src\org\crazyit\gamehall\server\ServerThread.java

```
private Socket socket;
public ServerThread(Socket socket) {
    this.socket = socket;
}
```

**ServerThread** 类继承 **Thread** 线程类，还需要重写 **Thread** 的 **run** 方法，在 **run** 方法中，我们需要将根据该 **Socket** 所得到的信息（客户端发送的信息）转化为具体的某个 **Request** 对象，服务器的线程得到 **Request** 对象后，就可以实例化 **Request** 中所包含的服务器处理类。

**ServerThread** 中的 **run** 方法。

代码清单：code\GameHall-Server\src\org\crazyit\gamehall\server\ServerThread.java

```
this.br = new BufferedReader(new InputStreamReader(this.socket.getInputStream()));
while((this.line = br.readLine()) != null) {
    //得到请求对象
    Request request = getRequest(this.line);
    //从 request 中得到客户端处理类, 并且构造 Response 对象
    Response response = new Response(request.getClientActionClass());
    //将请求的参数都设置到 Response 中
    copyParameters(request, response);
    //得到 Server 处理类
    ServerAction action = getAction(request.getServerActionClass());
    action.execute(request, response, this.socket);
}
```

以上代码中的 **getRequest** 方法，只要通过 **XStreamUtil** 中的 **toObject** 方法即可以将客户请求的字

字符串转换成一个 **Request** 对象, **XStreamUtil** 已经在 15.2.1 中实现。当服务器线程得到 **Request** 对象后, 就可以新建一个 **Response** 对象作为服务器的响应, 并根据 **Request** 对象中的 **serverActionClass** 属性得到具体的某个服务器处理 **Action** 类, 执行 **execute** 方法。需要特别注意的是, 通过 **Request** 的 **serverActionClass** 属性得到某个实现类的时候, 可以将得到的实现类实例放到一个 **Map** 中进行保存, 那么当用户多次请求同一个 **Action** 的时候, 就可以不再需要重复创建。

到这里, 我们框架的服务器端已经全部实现了, 在本章的代码中, 对应的是 **GameHall-Server** 模块, 按照以上的代码实现了该模块后, 这个服务器模块就可以不再需要作改动, 即使加入新的游戏, 它也可以不用进行代码改变。**GameHall** 模块只是服务器端负责处理转发的一个中间角色, 它不负责处理任何的业务逻辑。由于在 **ServerThread** 中, 我们需要使用反射来得到某个服务器处理类, 因此, 当加入一些新的游戏时, 我们就需要将这些游戏的模块加入到环境变量中。一些公用的接口或者类, 我们可以使用一个 **GameHall-Commons** 的模块来存放, 例如可以将 **Request**、**Response**、**ServerClass**、**ClientClass**、**User** 和 **Game** 等接口与类放到 **GameHall-Commons** 模块中, 客户端与服务器端的模块都依赖于 **GameHall-Commons** 模块。

### 15.2.4 编写框架客户端

由于我们这个游戏大厅的框架, 并不需要编写任何的逻辑, 因此客户端的实现与服务器端的实现基本类似。当用户编写了一个新的游戏时, 就可以直接将游戏的包放置到客户端的目录中, 用于给客户端加载相应的类。与服务器端的实现一样, 都是负责一个中转的功能, 当接收到服务器响应时 (得到 **Response** 对象的 XML), 就直接根据 **Response** 对象中的 **actionClass** 来得到客户端处理类, 同样再使用反射来得到某个客户端处理类的实例, 再调用 **execute** 方法。这样做, 就规定了发送请求时, 需要声明客户端处理类, 也就是设置 **Request** 对象中的 **clientActionClass**, 而在服务器处理时, 就会将这个属性设置为 **Response** 的 **actionClass**。每个客户端处理类都必须实现 **ClientAction** 接口。

编写客户端线程类 **ClientThread**, 该类的主体代码如下。

代码清单: code\GameHall-Client\src\org\crazyit\gamehall\client\ClientThread.java

```
InputStream is = this.user.getSocket().getInputStream();
BufferedReader br = new BufferedReader(new InputStreamReader(is));
while ((this.line = br.readLine()) != null) {
    Response response = getResponse(this.line);
    //得到客户端的处理类
    ClientAction action = getClientAction(response.getActionClass());
    //执行客户端处理类
    action.execute(response);
}
```

根据服务器返回的响应字符串, 将这些字符串转换成一个 **Response** 对象, 再通过该对象得到具体某个 **ClientAction** 的实现类, 再调用 **execute** 方法。与服务器端的实现一样, 客户端也并不需要处理任何的业务, 这些业务都交由客户端或者服务器端处理类进行。在本章中客户端模块为 **GameHall-Client**。

### 15.2.5 建立登录界面

框架的客户端除了提供一个客户端线程类之外, 还需要提供一个登录界面, 让用户选择进入游戏大厅 (某个游戏的大厅) 和输入相关的登录信息。由于登录界面并不与某个特定的游戏相关, 只是用户输入信息的一个界面, 因此可以将登录界面的相关类放到客户端的模块 **GameHall-Client** 中。登录界面如图 15.1 所示。



图 15.1 登录界面

在本章中，登录界面是 **GameHall-Client** 的 **LoginFrame** 类。登录界面需要注意的是头像的下拉框实现，需要到特定的某个目录中读取所有的头像文件，该目录存在于 **GameHall-Client** 模块。在读取头像文件的时候，我们需要将读取到的头像文件封装成一个 **Map** 对象，该对象里面 **key** 为头像图片的相对路径，**value** 就是该图片的 **ImageIcon** 对象。为了让下拉框中能显示图片，需要为下拉框 **JComboBox** 提供一个 **ListCellRenderer** 的实现类。

代码清单：code\GameHall-Client\src\org\crazyit\gamehall\client\HeadComboBoxRenderer.java

```
public Component getListCellRendererComponent(JList list, Object value,
    int index, boolean isSelected, boolean cellHasFocus) {
    String selectValue = (String)value;
    //设置背景颜色
    if (isSelected) {
        setBackground(list.getSelectionBackground());
        setForeground(list.getSelectionForeground());
    } else {
        setBackground(list.getBackground());
        setForeground(list.getForeground());
    }
    //从头像的 Map 中得到当前选择的头像图片
    Icon icon = this.heads.get(selectValue);
    setIcon(icon);
    if (icon != null) setFont(list.getFont());
    return this;
}
```

注意以上代码中，需要得到下拉框所选中的值，这个值必须是 **Map** 中的某个 **key**，这样的话，意味着需要将头像文件的相对路径作为值来创建一个 **JComboBox**。以下方法创建一个 **JComboBox** 对象。

代码清单：code\GameHall-Client\src\org\crazyit\gamehall\client>LoginFrame.java

```
//创建头像选择下拉
private void buildHeadSelect() {
    this.headSelect = new JComboBox(this.heads.keySet().toArray());
    this.headSelect.setMaximumRowCount(5);
    this.headSelect.setRenderer(new HeadComboBoxRenderer(this.heads));
}
```

```
}
```

以上的黑体代码，使用头像 Map 的 key 来创建 JComboBox 的选择项。除了头像的选择外，还需要特别注意的是游戏的选择下拉。在本章中，我们已经确定了游戏大厅只是一个简单的框架，在登录界面需要到某个目录去读取一些相关的包，将这些包所代表的游戏显示到游戏下拉中。因此还需要编写读取包信息的代码。

以下代码读取 jar 包。

代码清单：code\GameHall-Client\src\org\crazyit\gamehall\client\LoginFrame.java

```
File folder = new File("game");
for (File file : folder.listFiles()) {
    if (isJar(file.getName())) {
        //得到 jar 文件
        JarFile jar = new JarFile(file);
        //得到元数据文件
        Manifest mf = jar.getManifest();
        //获得各个属性
        Attributes gameClassAttrs = mf.getMainAttributes();
        //查找 Game-Class 属性
        String gameClass = gameClassAttrs.getValue("Game-Class");
        if (gameClass != null) {
            Game game = (Game)Class.forName(gameClass).newInstance();
            this.gameSelect.addItem(game);
        }
    }
}
```

以上的代码，到 GameHall-Client 中的 game 目录读取所有的 jar 包，再读取各个元数据文件（MANIFEST.MF 文件），得到里面声明的 Game-Class 属性，再通过反射将这个属性值转换成具体的某个 Game 的实现类。这个 Game-Class 属性的值就是该游戏的入口类。这样做无形之中为我们框架所加载的包加入了限制，如果是一个游戏客户端的包，就必须在 MANIFEST.MF 文件中声明 Game-Class 属性，这个属性定义该游戏的入口类。

到这里，游戏大厅的登录界面已经实现，下面小节实现玩家的登录功能。

### 15.2.6 实现登录功能

玩家输入了登录的相关信息后，就可以点击进行登录，在进行登录时，需要得游戏选择下拉框的值（某个游戏的入口类），并将当前的玩家信息封装成一个 User 对象，作为参数调用游戏入口类（Game 的实现类）的 start 方法。

以下为 LoginFrame 中的 login 方法。

代码清单：code\GameHall-Client\src\org\crazyit\gamehall\client\LoginFrame.java

```
//创建 Socket 并为 User 对象设置 Socket
this.user.setSocket(createSocket(this.connectionField.getText(), 12000));
//得到用户所选择的 game
Game game = (Game)this.gameSelect.getSelectedItem();
game.start(this.user);
//启动线程
ClientThread thread = new ClientThread(this.user);
thread.start();
this.setVisible(false);
```

以上的代码实现了登录功能，从界面的游戏下拉框中得到具体的某个游戏，由于游戏选择下拉框是

使用具体的某个游戏类来创建的，因此得到该类后，直接强制转换成 **Game**，再执行 **start** 方法并启动客户端线程。

到此，游戏大厅的基本框架已经完成，我们可以在这个框架的基础上开发各种游戏，但是前提是必须遵守该框架定义的一些规则。这些规则包括：

- ❑ 每个游戏都必须提供一个游戏入口类（**Game** 接口的实现类）；
  - ❑ 每个游戏打成 jar 包后，都需要在 **MANIFEST.MF** 文件中加入 **Game-Class** 属性来声明游戏入口类；
  - ❑ 客户端向服务器发送信息时，必须是代表一个 **Request** 对象的 XML 字符串；
  - ❑ 服务器向客户端发送信息时，必须是代表一个 **Response** 对象的 XML 字符串。
- 编写完最基本的框架后，下面章节编写一个五子棋游戏大厅。

## 15.3 建立五子棋游戏大厅

我们已经在 15.2 中编写了游戏大厅的基本框架，那么本小节开始编写一个五子棋的游戏大厅。在编写游戏大厅框架的时候，就定下规则，必须为游戏建立一个入口类。另外，五子棋的游戏大厅中还包括一系列的对象，包括桌子、位置等。我们将这些对象放到一个 **fivechess-commons** 的模块中。由于这些对象可能在客户端，也可能在服务器端使用到，因此我们将这些对象提取出来放到一个公共的模块中。

### 15.3.1 编写游戏大厅的对象

游戏大厅中最基本的单位就是桌子，但是每种游戏的游戏大厅都不一致，例如斗地主游戏大厅中的桌子可能会有四个座位，而五子棋只需要两个座位，而且每个游戏都会有自己的玩家对象，这些玩家对象都包括不同的内容。新建一个 **ChessUser** 类，继承 **User** 类，该类代表一个五子棋玩家对象。

**ChessUser** 包括以下属性：

```
//是否已经准备游戏
private boolean ready;
```

由于五子棋中每个桌子只有两个座位，因此我们为五子棋游戏新建一个 **Table** 对象来表示这个游戏的桌子对象。一个桌子中有两个座位，我们建立 **Seat** 对象来表示具体的某个座位。

**Seat** 对象包括如下属性。

代码清单：code\fivechess-commons\src\org\crazyit\gamehall\fivechess\commons\Seat.java

```
//该座位的玩家
private ChessUser user;
//座位的座标范围
private Rectangle range;
//座位边，只能为本类的 LEFT 和 RIGHT 属性值
private String side;
//座位宽度
public final static int SEAT_WIDTH = 30;
//座位高度
public final static int SEAT_HEIGHT = 30;
public final static String LEFT = "left";
public final static String RIGHT = "right";
```

**Table** 对象包括如下属性。

代码清单：code\fivechess-commons\src\org\crazyit\gamehall\fivechess\commons\Table.java

```
//桌子图片在大厅中的开始 X 座标
```

```

private int beginX;
//桌子图片在大厅中的开始 Y 座标
private int beginY;
//桌子的图片
private String tableImage;
//桌子号
private int tableNumber;
//默认的图片宽
public final static int DEFAULT_IMAGE_WIDTH = 140;
//默认的图片高
public final static int DEFAULT_IMAGE_HEIGHT = 140;
//该 Table 对应的范围
private Rectangle range;
//左边的座位
private Seat leftSeat;
//右边的座位
private Seat rightSeat;

```

这样，我们就定义了桌子和位置的对象，这里需要注意的是，位置并不需要知道自己属于哪张桌子，桌子对象中则提供了两个座位对象：**leftSeat** 和 **rightSeat**，表示一张桌子中只能有两个位置。**Table** 与 **Seat** 对象都提供了 **range** 属性，表示桌子或者位置的图片在大厅中具体位置范围。

为 **Table** 对象提供一个构造器，在创建 **Table** 的时候，就需要同时创建左边与右边的座位对象，并且设置相应的坐标位置。

**Table** 的构造器。

代码清单：code\fivechess-commons\src\org\crazyit\gamehall\fivechess\commons\Table.java

```

//创建桌子对象的时候就创建左右的 Seat 对象
this.leftSeat = new Seat(null, new Rectangle(getLeftSeatBeginX(), getLeftSeatBeginY(),
    Seat.SEAT_WIDTH, Seat.SEAT_HEIGHT), Seat.LEFT);
this.rightSeat = new Seat(null, new Rectangle(getRightSeatBeginX(), getRightSeatBeginY(),
    Seat.SEAT_WIDTH, Seat.SEAT_HEIGHT), Seat.RIGHT);

```

**Seat** 对象的构造器。

代码清单：code\fivechess-commons\src\org\crazyit\gamehall\fivechess\commons\Seat.java

```

public Seat(ChessUser user, Rectangle range, String side)

```

这里需要注意的是，位置的具体位置由它所在的桌子确定。

### 15.3.2 服务器创建游戏大厅数组

确定了游戏大厅的几个对象后，我们就可以在服务器中创建游戏大厅的数组，五子棋游戏的服务器端在本章中使用的是 **fivechess-server** 模块，游戏大厅的几个对象都保存在 **fivechess-commons** 模块，因此 **fivechess-server** 模块会依赖于 **fivechess-commons** 模块。游戏大厅数组保存在服务器端，我们使用一个 **ChessContext** 的类来保存这些信息。

代码清单：code\fivechess-server\src\org\crazyit\gamehall\fivechess\server\ChessContext.java

```

//保存桌子信息
public static Table[][] tables = new Table[TABLE_COLUMN_SIZE][TABLE_ROW_SIZE];
static {
    //初始化桌子信息
    tables = new Table[TABLE_COLUMN_SIZE][TABLE_ROW_SIZE];
    int tableNumber = 0;
    for (int i = 0; i < tables.length; i++) {
        for (int j = 0; j < tables[i].length; j++) {

```



```

        Table table = new Table(Table.DEFAULT_IMAGE_WIDTH*i,
                                Table.DEFAULT_IMAGE_HEIGHT*j, tableNumber);
        tables[i][j] = table;
        tableNumber++;
    }
}

```

建立一个 **Table** 的二维数组，并在 **ChessContext** 类的初始化块中建立这个数组。就这样，在服务端就保存了一个桌子的二维数组，当有新的玩家进入五子棋游戏大厅的时候，就将这个桌子的二维数组设置到 **Response** 的数据列表中，返回给所有的用户。对象与 XML 字符串进行互转的时候，尽量避免使用一些大对象，例如 **Image** 对象等，否则将对象转换成 XML 的时候，将生成大量的 XML 字符串，影响传输性能。

### 15.3.3 玩家进入游戏大厅

玩家通过登录界面，选择具体进入的某个游戏，就首先调用 **Game** 实现类的 **start** 方法。我们新建一个 **fivechess-client** 的模块，表示五子棋游戏大厅的客户端。按照之前的规则，提供游戏客户端，需要在 **MANIFEST.MF** 文件中声明一个 **Game-Class** 的属性，表示该游戏的入口类。下面为五子棋的游戏客户端提供一个入口类，实现 **Game** 接口。

代码清单：code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\action\ChessGame.java

```

public void start(User user) {
    //得到进入游戏的玩家信息
    ChessUser cu = convertUser(user);
    ChessClientContext.chessUser = cu;
    //构造一次请求，告诉服务器玩家进入大厅，服务器响应处理类是 LoginAction
    Request req = new Request("org.crazyit.gamehall.fivechess.server.action.LoginAction",
                              "org.crazyit.gamehall.fivechess.client.action.ClientInAction");
    req.setParameter("user", cu);
    //得到玩家的 Socket 并发送请求，告诉服务器玩家进入了大厅
    cu.getPrintStream().println(XStreamUtil.toXML(req));
}

```

**ChessGame** 的 **start** 方法，首先将玩家对象 (**User**) 对象转换成一个五子棋玩家对象 (**ChessUser**)，再将当前的五子棋玩家对象设置到五子棋游戏的上下文对象中，最后，构造一次请求，发送到服务器，告诉服务器，新的玩家进入了五子棋游戏大厅。在以上代码中，声明了服务器处理类是服务器端的 **LoginAction**，客户端处理类是 **ClientInAction**。

当玩家发送了请求到服务器后，最先处理请求的是 **Game-Server** 模块，该模块将玩家的请求转发到具体的某个 **ServerAction** 实现类中，这里的 **LoginAction** 就是我们的服务器处理类。这里需要注意的是 **LoginAction** 位置 **fivechess-server** 模块中，表示该类是属于服务器执行的类。在编写 **LoginAction** 前，我们需要明白这 **Action** 需要进行的一些动作，玩家进入游戏大厅，首先必须将玩家的信息保存到服务器中，再将游戏大厅当前所有的信息发送给登录的玩家。

代码清单：

code\fivechess-server\src\org\crazyit\gamehall\fivechess\server\action>LoginAction.java

```

public void execute(Request request, Response response, Socket socket) {
    //从请求参数中得到 ChessUser
    ChessUser cu = (ChessUser)request.getParameter("user");
    cu.setSocket(socket);
    //加入到所有玩家中
    ChessContext.users.put(cu.getId(), cu);
}

```

```
//将玩家设置到响应中
response.setData("user", cu);
//将所有玩家信息设置到响应中
response.setData("users", ChessContext.users);
//将所有的桌子信息返回到客户端
response.setData("tables", ChessContext.tables);
//将大厅中桌子的列数和行数返回到客户端
response.setData("tableColumnSize", ChessContext.TABLE_COLUMN_SIZE);
response.setData("tableRowSize", ChessContext.TABLE_ROW_SIZE);
//返回给登录玩家, 登录成功
cu.getPrintStream().println(XStreamUtil.toXML(response));
}
```

我们将进入游戏大厅的玩家保存到服务器上下文中, 因此需要在 **ChessContext** 中添加一个属性来保存玩家信息:

```
public static Map<String, ChessUser> users = new HashMap<String, ChessUser>();
```

在服务器上下文中使用一个 **Map** 来保存玩家信息, 这个 **Map** 的 **key** 是玩家对象的 **id**, **value** 是具体的某个五子棋玩家对象。玩家通过 **ChessGame** 来发送第一次请求给服务器, **LoginAction** 负责处理这一次请求, 然后将 **Response** 对象转换成 **XML** 字符串返回给客户端, 接下来, 就是客户端处理类 (**ClientInAction**) 负责处理这一次服务器响应。服务器响应首先是发送给 **fivechess-client** 模块的 **ClientThread** 类进行处理, 该类同样地负责转发, 去寻找具体的某个客户端处理类 (**ClientAction**) 的实现类进行处理。**ClientInAction** 接收到服务器的响应后, 就需要为刚登录的玩家创建游戏大厅, 服务器的响应中已经包括创建游戏大厅所需的各个信息, 包括桌子, 玩家等。我们暂时不提供实现, 下面创建游戏大厅的各个界面。

### 15.3.4 创建游戏大厅界面

游戏大厅界面在本章对应的是 **GameHallFrame** 类, 该类在 **fivechess-client** 模块中。我们需要做的效果如图 15.2 所示。

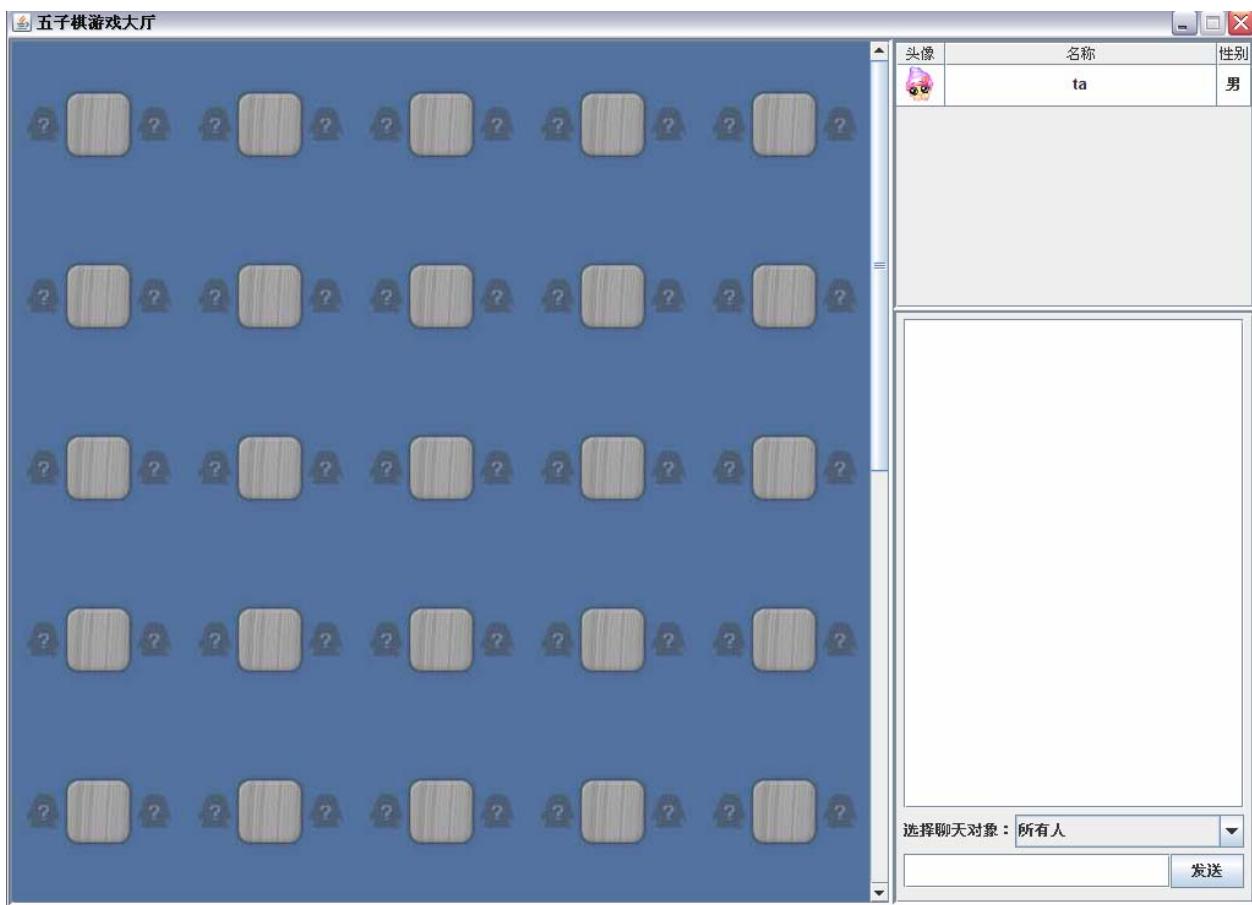


图 15.2 游戏大厅界面

五子棋游戏大厅主要包括一个存放桌子的 `JPanel`，存放所有玩家的 `JTable` 对象，主要用于聊天的 `JPanel`，玩家列表的 `JTable` 对象与聊天对象界面较为简单，复杂的是大厅对象，该对象在本章中对应的 `HallPanel` 对象。

当登录进入五子棋游戏大厅的时候，我们首先会调用 `ChessGame` 的 `start` 方法，该方法会发送一次请求到服务器取全部的桌子信息，再由客户端的 `ClientInAction` 负责创建游戏界面。在服务器响应中我们可以得到所有的桌子信息，然后再根据这些桌子的信息去创建 `HallPanel`。

以下是 `HallPanel` 的 `paint` 方法，主要用于绘画界面中的桌子与玩家。

代码清单：`code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\HallPanel.java`

```
public void paint(Graphics g) {
    for (int i = 0; i < this.tables.length; i++) {
        for (int j = 0; j < this.tables[i].length; j++) {
            Table table = this.tables[i][j];
            Seat leftSeat = table.getLeftSeat();
            Seat rightSeat = table.getRightSeat();
            //画桌子的图片
            g.drawImage(tableImage, table.getBeginX(),
                table.getBeginY(), this);
            //画左边座位的玩家
            if (leftSeat.getUser() != null) {
                Image head = getHeadImage(leftSeat.getUser().getHeadImage());
                g.drawImage(head, table.getLeftSeatBeginX(),
                    table.getLeftSeatBeginY(), this);
            }
        }
    }
}
```

```

    }
    //画右边座位的玩家
    if (rightSeat.getUser() != null) {
        Image head = getHeadImage(rightSeat.getUser().getHeadImage());
        g.drawImage(head, table.getRightSeatBeginX(),
            table.getRightSeatBeginY(), this);
    }
}
}
}

```

**HallPanel** 得到各个桌子的二维数组后, 根据这个二维数组去绘画桌子, 如果桌子中的座位有玩家的话, 就画上相应的玩家头像。画完桌子后, 我们需要处理鼠标事件, 当鼠标的指针移动到某个位置上面的时候, 就需要帮这个位置更换图片, 让其做到有阴影的效果。

为鼠标移动添加事件。

代码清单: code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\HallPanel.java

```

this.addMouseMotionListener(new MouseMotionAdapter() {
    public void mouseMoved(MouseEvent e) {
        moveMouse(e);
    }
});

```

玩家移动鼠标的时候, 就触发 **mouseMove** 方法, 该方法的主体代码如下。

代码清单: code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\HallPanel.java

```

if (table.getLeftSeat().getRange().contains(x, y)) {
    //左边座位
    this.setCursor(HAND_CURSOR);
    //如果位置上没有人才更换图片
    if (table.getLeftSeat().getUser() == null) {
        g.drawImage(seatSelectImage, table.getLeftSeatBeginX(),
            table.getLeftSeatBeginY(), this);
    }
} else if (table.getRightSeat().getRange().contains(x, y)) {
    //右边座位
    this.setCursor(HAND_CURSOR);
    //如果位置上没有人才更换图片
    if (table.getRightSeat().getUser() == null) {
        g.drawImage(seatSelectImage, table.getRightSeatBeginX(),
            table.getRightSeatBeginY(), this);
    }
} else {
    this.setCursor(DEFAULT_CURSOR);
    this.repaint();
}
}

```

当鼠标移动到有人的位置时, 就不更改阴影图片, 如果鼠标移动到没有人的座位时, 才更换座位的阴影图片与鼠标指针, 具体的效果如图 15.3 所示。



15.3 鼠标指针移动到空座位上时

### 15.3.5 创建玩家列表与聊天界面

玩家登录进入游戏大厅的时候，服务器需要将所有玩有的信息发送给登录的玩家，界面得到这些玩这有信息后，就将所有的玩家设置到玩家列表中。在本章，玩家列表使用的是 **UserTable** 来表示。**UserTable** 的实现较为简单，只需要得到具体的五子棋玩家列表，就可以根据这些玩家来创建列表。该列表需要注意的是，列表的单元格并不可以编辑。需要将自己放到所有玩家的最前面。

由于玩家列表中涉及图片的显示，同样也是需要提供一个 **DefaultTableCellRenderer** 来显示相应列的图片。编写一个 **DefaultTableCellRenderer** 的子类来处理显示头像图片：

代码清单：

code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\UserTableCellRenderer.java

```
public Component getTableCellRendererComponent(JTable table, Object value,
        boolean isSelected, boolean hasFocus, int row, int column) {
    setHorizontalAlignment(SwingConstants.CENTER);
    //设置显示图片
    if (value instanceof Icon) this.setIcon((Icon)value);
    else this.setText(value.toString());
    //设置单元格的背景颜色
    if (isSelected) this.setBackground(Color.YELLOW);
    else this.setBackground(Color.WHITE);
    return this;
}
```

最后将各个玩家的信息转换成列表显示的数据类型显示到列表中，在 **UserTable** 中，我们使用一个 **List** 来保存所有的玩家信息，提供一个 **getDatas** 的方法来转换玩家集合。

代码清单：code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\UserTable.java

```
//得到玩家列表的数据格式
private Vector<Vector> getDatas() {
    Vector<Vector> result = new Vector<Vector>();
    for (ChessUser user : this.users) {
        Vector v = new Vector();
        v.add(user.getId());
        v.add(getHead(user.getHeadImage()));
        v.add(user.getName());
        v.add(getSex(user.getSex()));
        result.add(v);
    }
    return result;
}
```

如图 15.2 所示，除了玩家列表外，还有聊天的界面，聊天界面在本章中使用 **ChatPanel** 表示。在 **ChatPanel** 中，只需要创建一些基本的界面组件即可，包括一个 **JTextArea**、一个 **JTextField** 和一个发送的按钮。需要游戏的是，构建 **ChatPanel** 时，也需要将所有的玩家与当前的玩家都作为构造参数传入，这是由于如果进行聊天，就需要让玩家选择聊天的对象，再发送给服务器。

玩家列表与聊天界面已经创建完成，这两个界面组件可以在游戏界面重用，游戏界面的实现将在下面章节讲述。

### 15.3.6 使用服务器的数据创建游戏大厅

建立游戏大厅的各个界面，都离不开玩家的信息。在 15.3.3 中，我们还没有实现 `ClientInAction`，这个客户端处理类主要用于得到服务器传过来的大厅信息、玩家信息，我们可以根据这些信息来创建游戏大厅的界面。

代码清单：

code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\action\ClientInAction.java

```
public void execute(Response response) {
    //从服务器中得到大厅信息并封装成一个 GameHallInfo 对象
    GameHallInfo hallInfo = getGameHallInfo(response);
    //得到全部玩家的信息
    List<ChessUser> users = getUsers(response);
    //得到进入游戏的玩家信息
    ChessUser cu = ChessClientContext.chessUser;
    //创建界面 GameHallFrame
    GameHallFrame mainFrame = new GameHallFrame(hallInfo, cu, users);
    mainFrame.sendUserIn();
}
```

以上代码的 `getGameHallInfo` 方法与 `getUsers` 方法，从服务器响应中得到桌子的信息与所有用户的信息，由于服务器处理类在接收用户进入游戏大厅信息的时候，就已经将这些信息设置到服务器响应里，因此只需要在客户端处理类中通过 `getDatas(key)` 方法就可以得到这些数据，这些数据在 15.3.3 中的 `LoginAction` 中已经设置。

客户端将接收到的数据创建游戏大厅后，还需要调用游戏大厅对象（`GameHallFrame`）的 `sendUserIn` 方法，这个方法主要用于客户端发送请求到服务器，告诉服务器，当前的玩家已经成功进入游戏大厅了。以下是 `sendUserIn` 方法的主要代码。

代码清单：code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\GameHallFrame.java

```
//构造一次请求，告诉服务器用户进入大厅，服务器响应处理类是 ReceiveInAction
Request req = new Request("org.crazyit.gamehall.fivechess.server.action.NewUserInAction",
    "org.crazyit.gamehall.fivechess.client.action.ReceiveInAction");
req.setParameter("userId", this.user.getId());
//得到用户的 Socket 并发送请求，告诉服务器用户进入了大厅
this.user.getPrintStream().println(XStreamUtil.toXML(req));
```

`sendUserIn` 方法重新构造一次请求，并发送给服务器，服务器处理类是 `NewUserInAction`，返回给客户端处理类是 `ReceiveInAction`。`NewUserInAction` 是服务器接收新玩家进入游戏大厅的请求，然后根据这个请求得到相应的玩家 id，再告诉这个大厅中的其他玩家，有新的玩家进入了游戏大厅了。

代码清单：code\GameHall-Server\src\org\crazyit\gamehall\server\action\NewUserInAction.java

```
public void execute(Request request, Response response, Socket socket) {
    //得到新登录的玩家
    String userId = (String)request.getParameter("userId");
    ChessUser user = ChessContext.users.get(userId);
    //将新玩家信息放到响应中
    response.setData("newUser", user);
    //向所有玩家发送信息
    for (String id : ChessContext.users.keySet()) {
        ChessUser hasLogin = ChessContext.users.get(id);
        //不必发送给自己
    }
```

```
        if (id.equals(user.getId())) continue;
        hasLogin.getPrintStream().println(XStreamUtil.toXML(response));
    }
}
```

**NewUserInAction** 放在 **fivechess-server** 模块中，表示该 **Action** 由服务器执行。下面为客户端创建一个 **ReceiveInAction**，用于客户端接收服务器发送“有新玩家进入”的信息。**ReceiveInAction** 是客户端执行的 **Action**。

代码清单：

code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\action\ReceiveInAction.java

```
public void execute(Response response) {
    //得到新进入的玩家
    ChessUser newUser = (ChessUser)response.getData("newUser");
    //向玩家列表中加入一个新玩家
    UserTable userTable = (UserTable)UIContext.modules.get(UIContext.HALL_USER_TABLE);
    userTable.addUser(newUser);
    //向聊天内容中添加
    ChatPanel chatPanel = (ChatPanel)UIContext.modules.get(UIContext.HALL_CHAT_PANEL);
    chatPanel.appendContent(newUser.getName() + " 进来了");
    chatPanel.refreshJComboBox();
}
```

以上代码需要注意的是，新建一个 **UIContext** 来保存各个界面组件，**UIContext** 中提供一个 **Map** 对象来保存界面组件，当有新的界面组件被创建时，就需要加入到 **Map** 中，并为该组件提供一个唯一的名称。

**ReceiveInAction** 是玩家用玩接收其他玩家进入游戏大厅的消息，一旦有新的玩家进入，服务器就会向所有玩家发送消息，有新的玩家进入，需要更新玩家列表，更新聊天界面组件的下拉框，最后在 **ChatPanel** 添加消息提示。具体的效果如图 15.4 所示。

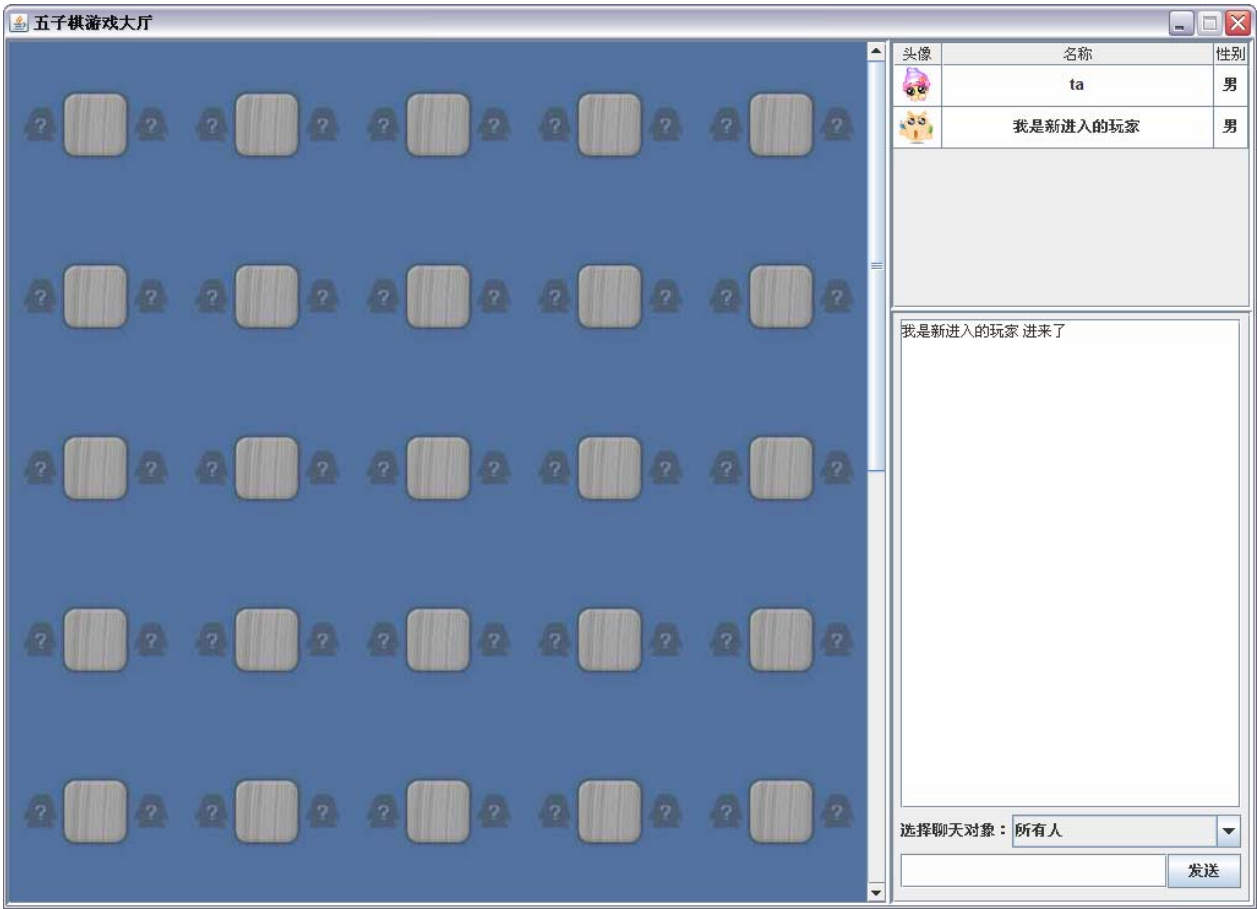


图 15.4 新的玩家进入游戏大厅

在本小节中，我们编写了游戏大厅的各个界面组件与对象，并实现了用户进入游戏大厅的功能，在下面章节，我们将实现游戏大厅的其他功能。

## 15.4 实现聊天功能

在游戏大厅中，我们提供了一个聊天的界面，玩家可以在上面进行聊天，发送或者接收聊天信息，需要注意的是，聊天界面是可以共用的，除了在游戏大厅中使用聊天界面外，还可以在游戏界面中使用。本小节实现游戏大厅中的聊天功能。

### 15.4.1 发送聊天信息

在聊天界面（ChatPanel）中，当玩家选择了某个聊天的对象，输入内容再点击发送按钮后，就可以创建一次请求，将该请求发送到服务器，告诉服务器：我对某个人（所有人）发送了聊天内容，请帮我转发。服务器得到这个请求后，就执行某个服务器处理类，该处理类就将聊天内容转发给相应的玩家。

以下是 ChatPanel 发送聊天内容的方法。

代码清单：code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\ChatPanel.java

```
//发送信息
public void send() {
    //得到发送的内容
```



```

String content = this.conentField.getText();
//得到接收玩家
ChessUser receiver = (ChessUser)this.target.getSelectedItem();
//构造请求
Request request = new Request(this.serverAction, this.clientAction);
//设置参数
request.setParameter("receiverId", receiver.getId());
request.setParameter("senderId", this.user.getId());
request.setParameter("content", content);
//发送请求
this.user.getPrintStream().println(XStreamUtil.toXML(request));
appendContent("你对 " + receiver.getName() + " 说: " + content);
}

```

以上的代码中，构造 **Request** 对象时，我们使用的是 **serverAction** 与 **clientAction** 这两个类属性，由于这个 **ChatPanel** 是可以重用的，因此 **serverAction** 与 **clientAction** 也是由使用者来提供。游戏大厅中发送聊天请求的服务器处理类是 **SendMessageAction**，客户端处理类是 **ReceiveMessageAction**。**Request** 中包括了接收人 id、发送人 id 和聊天内容的信息。服务器中的 **SendMessageAction** 得到这些信息后，就可以将内容发送给相关的玩家，以下是 **SendMessageAction** 的具体实现。玩家发送聊天信息，由服务器的 **SendMessageAction** 接收，再由该 **Action** 转发，让客户端的 **ReceiveMessageAction** 负责处理。

代码清单：

code\GameHall-Server\src\org\crazyit\gamehall\server\action\SendMessageAction.java

```

public void execute(Request request, Response response, Socket socket) {
    String receiverId = (String)request.getParameter("receiverId");
    String senderId = (String)request.getParameter("senderId");
    ChessUser sender = ChessContext.users.get(senderId);
    String content = (String)request.getParameter("content");
    if (receiverId == null) {
        //向所有人发
        for (String id : ChessContext.users.keySet()) {
            if (id.equals(senderId)) continue;
            ChessUser cu = ChessContext.users.get(id);
            response.setData("content", sender.getName() + " 对 所有人 说: " + content);
            cu.getPrintStream().println(XStreamUtil.toXML(response));
        }
    } else {
        //得到接收人
        ChessUser receiver = ChessContext.users.get(receiverId);
        if (receiver.getId().equals(sender.getId())) return;
        response.setData("content", sender.getName() + " 对你说: " + content);
        receiver.getPrintStream().println(XStreamUtil.toXML(response));
    }
}

```

**SendMessageAction** 中根据请求的参数来发送相应的信息，需要注意的是，如果接收人 id 为空的话，那么就意味着向所有人发送聊天内容。

### 15.4.2 接收聊天信息

接收聊天信息由 **ReceiveMessageAction** 负责处理，该类属于客户端处理类，在 **fivechess-client**

模块中，某个玩家得到聊天内容后，就可以获得界面组件，再将这些聊天内容追加到聊天界面组件的文本域中。

代码清单：

code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\action\ReceiveMessageAction.java

```
public void execute(Response response) {  
    //得到聊天的界面组件  
    ChatPanel chatPanel = (ChatPanel)UIContext.modules.get(UIContext.HALL_CHAT_PANEL);  
    //从服务器响应中得到内容  
    String content = (String)response.getData("content");  
    chatPanel.appendContent(content);  
}
```

聊天的具体效果如图 15.5 所示。

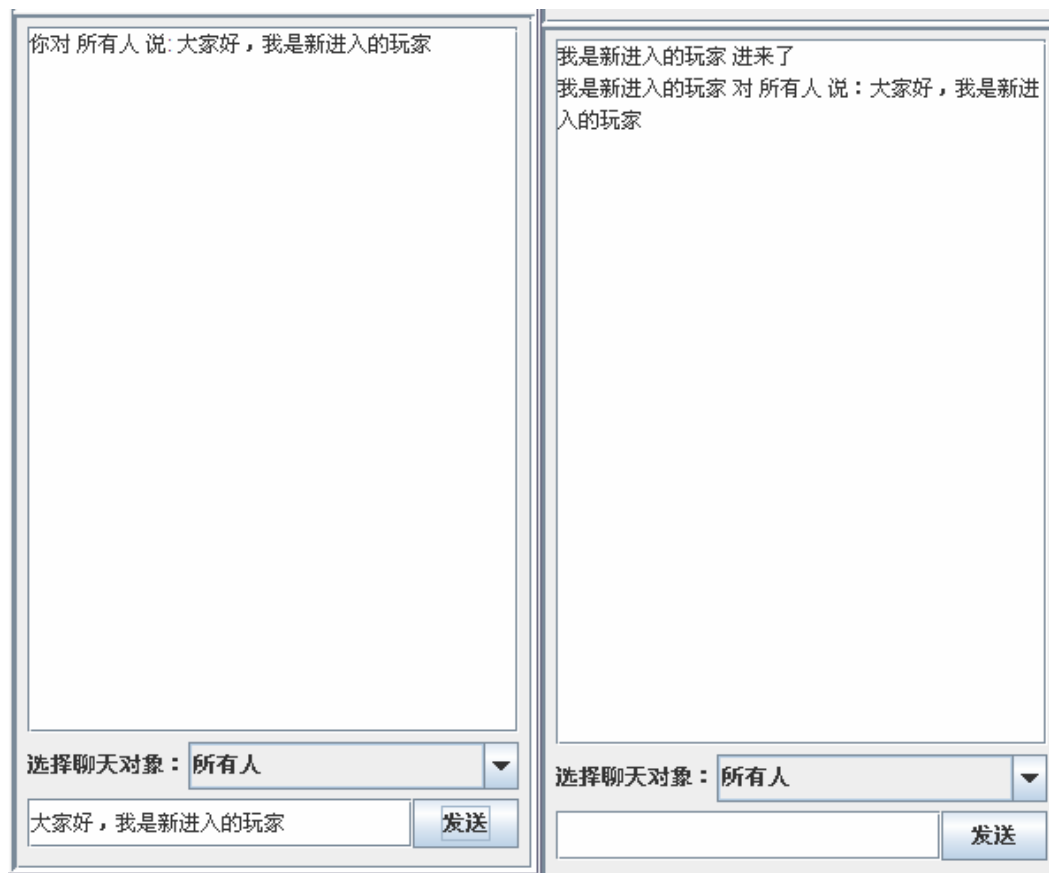


图 15.5 聊天效果

## 15.5 启动游戏

玩家选择了某一个位置坐下的时候，需要对位置进行判断，看下该位置是否有人，还要对玩家当前的状态进行判断，判断其是否已经坐下了，如果玩家没有坐到任何位置上并且当前所选择的位置没有玩家，就可以坐到位置上并展现游戏界面。

### 15.5.1 建立游戏界面

游戏界面的窗口在本章中使用 **ChessFrame** 类，该类继承于 **JFrame**，**ChessFrame** 包括三个界面组件，一个是游戏区域的 **GamePanel**，另外两个就是我们已经实现的用户列表（**UserTable** 对象）和聊天界面对象（**ChatPanel**），这里主要实现 **GamePanel**。游戏界面的效果如图 15.6 所示。

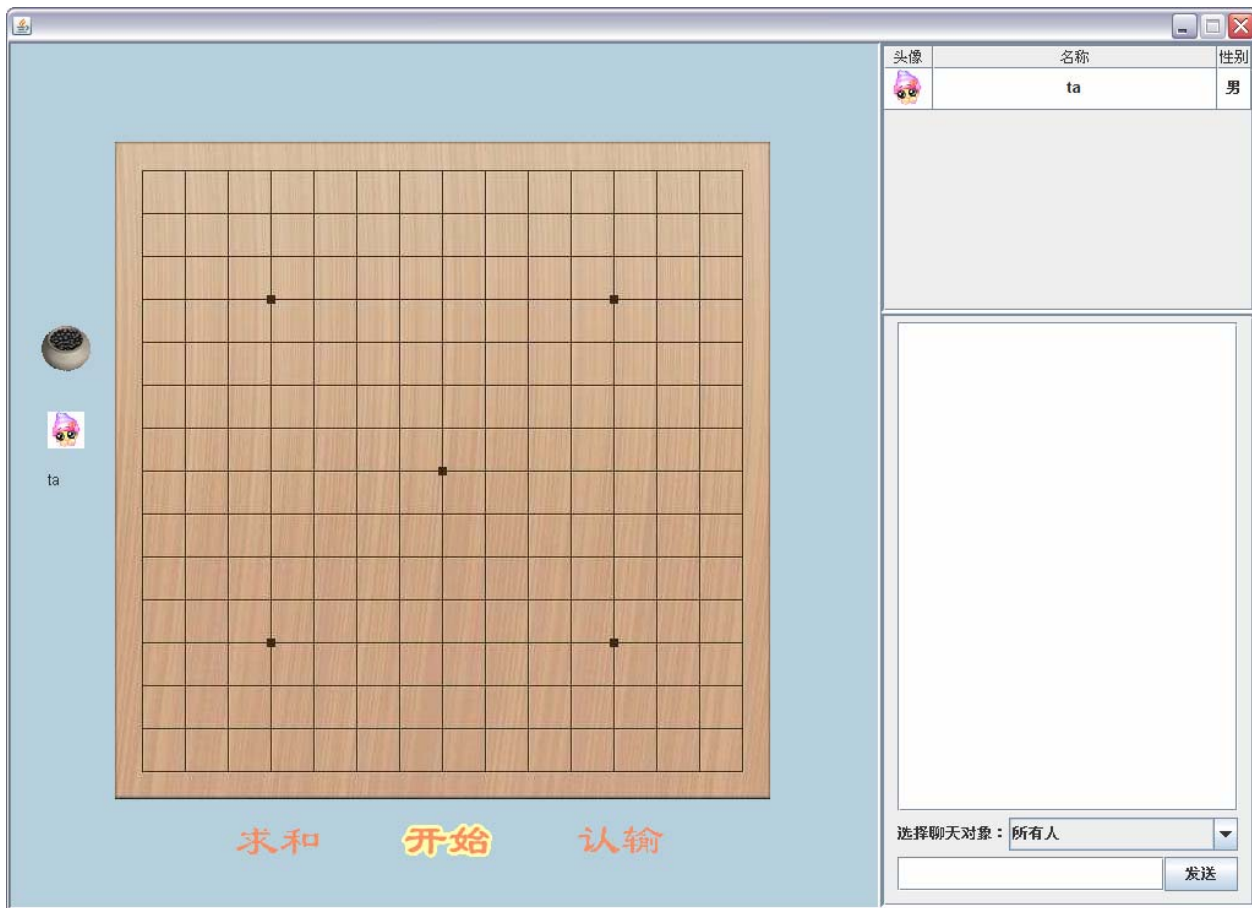


图 15.6 游戏界面

游戏区 (**GamePanel**) 只需要将棋盘图片、玩家头像、工具栏图片绘画出来即可，以下是 **GamePanel** 的 **paint** 方法的主体代码。

代码清单：

code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\game\GamePanel.java

```
g.drawImage(background, 0, 0, this.getWidth(), this.getHeight(), this);
g.drawImage(chessboard, 85, 80, this);
ChessUser lu = this.table.getLeftSeat().getUser();
ChessUser ru = this.table.getRightSeat().getUser();
//设置左边玩家的头像
this.leftUserHead = getUserHead(this.leftUserHead, lu);
//设置右边玩家的头像
this.rightUserHead = getUserHead(this.rightUserHead, ru);
//画头像
g.drawImage(this.leftUserHead, 30, 300, this);
g.drawImage(this.rightUserHead, 645, 300, this);
//画左边的玩家
```

```

drawLeftUser(g, lu);
//画右边的玩家
drawRightUser(g, ru);
//画工具栏
g.drawImage(this.currentToolImage, 160, 630, this);

```

在游戏区需要画的有棋盘图片、左边玩家的头像图片与玩家名称、右边玩家的头像图片与玩家名称、工具栏图片。绘画这些图片与文件只需要注意各个界面元素的坐标，根据这些坐标值将相应的图片画到 `GamePanel` 中。创建一个 `GamePanel` 对象，需要得到具体的一个桌子对象和当前的玩家对象，以下是 `GamePanel` 的构造器的部分代码：

```

public GamePanel(Table table, ChessUser user) {
    this.table = table;
    this.user = user;
}

```

## 15.5.2 玩家坐下

玩家选择了具体的一个位置点击时，需要进行一系列的判断，判断位置是否有人，判断玩家是否已经坐在其他位置上，当条件都不成立时，就可以向服务器发送信息，告诉服务器当前的玩家已经坐到具体的某个位置上，再向游戏大厅中所有的玩家发送信息：该玩家已经坐在某张桌子中了。

以下是 `HallPanel` 的 `sitDown` 方法。

代码清单：code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\HallPanel.java

```

//坐下的桌子的方法
private void sitDown(MouseEvent e) {
    int x = e.getX();
    int y = e.getY();
    //获得桌子
    Table table = getTable(x, y);
    if (table != null) {
        //得到座位
        Seat seat = getSeat(table, x, y);
        if (seat != null) {
            //判断玩家是否已经坐下
            if (this.user.hasSitDown(this.tables)) {
                //可以提示玩家
                return;
            }
            if (seat.getUser() != null) {
                //座位上有人，可以提示玩家
                return;
            } else {
                seat.setUser(this.user);
                //没人向服务器发送请求
                sendServerSitDown(table, seat.getSide());
            }
        }
    }
}

```

做了一系列判断后，玩家就可以坐到位置上，得到该位置的对象（`Seat`）后，再设置该位置的玩家，`Seat` 的 `user` 属性。最后调用 `sendServerSitDown` 方法告诉服务器。

sendServerSitDown 方法。

代码清单: code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\HallPanel.java

```
//向服务器发送请求, 告诉服务器自己已经坐下
private void sendServerSitDown(Table table, String side) {
    //创建 Request 对象并设置相关的参数
    Request request = new Request("org.crazyit.gamehall.fivechess.server.action.UserSitDownAction",
        "org.crazyit.gamehall.fivechess.client.action.ReceiveUserSitDownAction");
    request.setParameter("tableNumber", table.getTableNumber());
    request.setParameter("side", side);
    request.setParameter("userId", this.user.getId());
    //设置启动游戏的类
    request.setParameter("beginClass",
        "org.crazyit.gamehall.fivechess.client.action.game.EnterGameAction");
    this.user.getPrintStream().println(XStreamUtil.toXML(request));
    this.repaint();
}
```

sendServerSitDown 方法构造一个 Request 对象, 向服务器发送玩家坐到位置上的信息, Request 需要保存桌子的桌子号、玩家 id 和座位位置的信息, 服务器端处理类是 UserSitDownAction。这里需要注意的是, 由于这次请求需要告诉其他玩家, 当前的玩家坐下了, 还需要告诉当前的玩家去启动游戏界面, 因此需要为 Request 再设置多一个参数值 beginClass, 表示当前玩家启动游戏界面的类, 这个 beginClass 是客户端处理类 EnterGameAction。

UserSitDownAction 的 execute 方法。

代码清单: code\GameHall-Server\src\org\crazyit\gamehall\server\action\UserSitDownAction.java

```
//得到桌子编号
Integer tableNumber = (Integer)request.getParameter("tableNumber");
String side = (String)request.getParameter("side");
Table table = Table.getTable(tableNumber, ChessContext.tables);
//得到刚坐下的玩家
String userId = (String)request.getParameter("userId");
ChessUser user = ChessContext.users.get(userId);
//得到座位
Seat seat = table.getSeat(side);
seat.setUser(user);
//告诉所有的客户端, 刚坐下的玩家在哪张桌子哪个位置坐下了
response.setData("tableNumber", table.getTableNumber());
response.setData("side", side);
response.setData("user", user.getId());
//向所有玩家发送信息, 有新玩家坐下
printResponse(user, response);
//得到启动游戏的客户端类
String beginClass = (String)request.getParameter("beginClass");
response.setActionClass(beginClass);
//告诉客户端, 需要启动游戏界面
user.getPrintStream().println(XStreamUtil.toXML(response));
```

UserSitDownAction 中的 execute 方法向客户端发送了两次服务器响应, 一次是告诉所有的玩家, 当前的玩家坐到某个位置上了, 第二次是告诉当前玩家, 需要启动游戏界面类。所有玩家接收到当前玩家坐下的信息后, 客户端处理类是 ReceiveUserSitDownAction, 当前玩家启动游戏界面的客户端处理类是 EnterGameAction。

代码清单:

code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\action\ReceiveUserSitDownAction.java

```
//得到界面对象
HallPanel hallPanel = (HallPanel)UIContext.modules.get(UIContext.HALL_PANEL);
//有新的玩家坐下, 这里由所有玩家(不包括发送人)执行
int tableNumber = (Integer)response.getData("tableNumber");
String side = (String)response.getData("side");
String userId = (String)response.getData("userId");
hallPanel.newUserSitDown(tableNumber, side, userId);
```

`ReceiveUserSitDownAction` 是客户端处理类, 接收其他玩家坐下了的信息, 最后调用游戏大树界面组件类 `HallPanel` 的 `newUserSitDown` 方法处理玩家坐下的事件。以下是 `HallPanel` 的 `newUserSitDown` 的具体实现。

代码清单: code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\HallPanel.java

```
//新玩家坐下, tableNumber 为桌子编号, side 为左右位置
public void newUserSitDown(int tableNumber, String side, String userId) {
    //得到桌子对象
    Table table = getTable(tableNumber);
    //得到座位
    Seat seat = table.getSeat(side);
    //得到坐下的玩家
    ChessUser newUser = getUser(userId);
    seat.setUser(newUser);
    this.repaint();
}
```

只要知道玩家坐在哪张桌子的哪一边, 就可以更新当前的 `Table` 二维数组, 再进行 `repaint`。这是 `ReceiveSitDownAction` 中向其他玩家发送的第一次服务器响应, 该 `Action` 向当前玩家发送第二次服务器响应时, 就需要改变 `Response` 的 `actionClass` 属性, 让发送坐下请求的玩家启动游戏界面 (`EnterGameAction`)。在这里需要注意的是, 如果当前玩家进入了游戏中, 需要判断当前玩家所坐下的桌子是否有人 (对手), 如果有对手的话, 就要告诉对方自己进来了, 这时也需要构造一个新的请求发送到服务器中, 让服务器去告诉当前玩家的对手。

代码清单:

code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\action\game\EnterGameAction.java

```
HallPanel hallPanel = (HallPanel)UIContext.modules.get(UIContext.HALL_PANEL);
//从服务器呼应中得到桌子编号
Integer tableNumber = (Integer)response.getData("tableNumber");
String side = (String)response.getData("side");
//根据桌子编号得到桌子信息
Table table = Table.getTable(tableNumber, hallPanel.getTables());
//显示界面
ChessFrame cf = new ChessFrame(table, ChessClientContext.chessUser);
cf.setVisible(true);
//告诉对方进入游戏(如果有对方玩家的话)
Seat seat = table.getSeat(side);
//得到对方座位
Seat otherSeat = table.getAnotherSeat(seat);
if (otherSeat.getUser() != null) {
    //有对手, 向服务器发送请求
    Request request = new Request(
        "org.crazyit.gamehall.fivechess.server.action.OpponentEnterAction",
```

```

        "org.crazyit.gamehall.fivechess.client.action.game.OpponentEnterAction");
//firstUserId 是对手的 ID(第一个进入游戏的玩家)
request.setParameter("firstUserId", otherSeat.getUser().getId());
//secondUserId 是自己的 ID(后进入游戏的玩家)
request.setParameter("secondUserId", seat.getUser().getId());
ChessClientContext.chessUser.getPrintStream().println(XStreamUtil.toXML(request));
    }

```

告诉对手自己进入了游戏，服务器处理类是 fivechess-server 的 OpponentEnterAction，对手执行的客户端处理类是 fivechess-client 的 OpponentEnterAction。

服务器端的 OpponentEnterAction。

代码清单：

code\GameHall-Server\src\org\crazyit\gamehall\server\action\OpponentEnterAction.java

```

//得到第一个玩家的对象(发送请求的人的对手)
String firstUserId = (String)request.getParameter("firstUserId");
ChessUser firstUser = ChessContext.users.get(firstUserId);
//得到第二个玩家的对象
String secondUserId = (String)request.getParameter("secondUserId");
response.setData("opponentId", secondUserId);
//告诉第一个进入游戏的玩家，有对手进入
firstUser.getPrintStream().println(XStreamUtil.toXML(response));

```

服务器端的 Action 主要根据参数找到发送这次请求的玩家的对手，告诉他当前玩家已经进入了游戏。因为服务器响应中就需要将当前玩家的信息设置到响应数据中，再找到当前玩家的对手，向其输入服务器响应。

客户端的 OpponentEnterAction。

代码清单：

code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\action\game\OpponentEnterAction.java

```

//得到大厅对象
HallPanel gameHall = (HallPanel)UIContext.modules.get(UIContext.HALL_PANEL);
//得到对手的 ChessUser 对象
String opponentId = (String)response.getData("opponentId");
//从大厅中得到对手的信息
ChessUser opponent = gameHall.getUser(opponentId);
ChessFrame cf = (ChessFrame)UIContext.modules.get(UIContext.GAME_FRAME);
cf.newUserIn(opponent);

```

客户的 OpponentEnterAction 主要用于处理接收对手进入我所坐的桌子的服务器响应，当我的对手进入桌子时，服务器就会告诉我，我的对手进来了，最后调用 ChessFrame 的 newUserIn 方法。

ChessFrame 的 newUserIn 方法。

代码清单：

code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\game\ChessFrame.java

```

//向玩家集合中添加一个新玩家，表示有新玩家进入
public void newUserIn(ChessUser user) {
    this.users.add(user);
    refreshUI();
}

```

newUserIn 方法向界面中的玩家集合添加一个新的玩家并更新界面组件。具体的效果如图 15.7 所示。

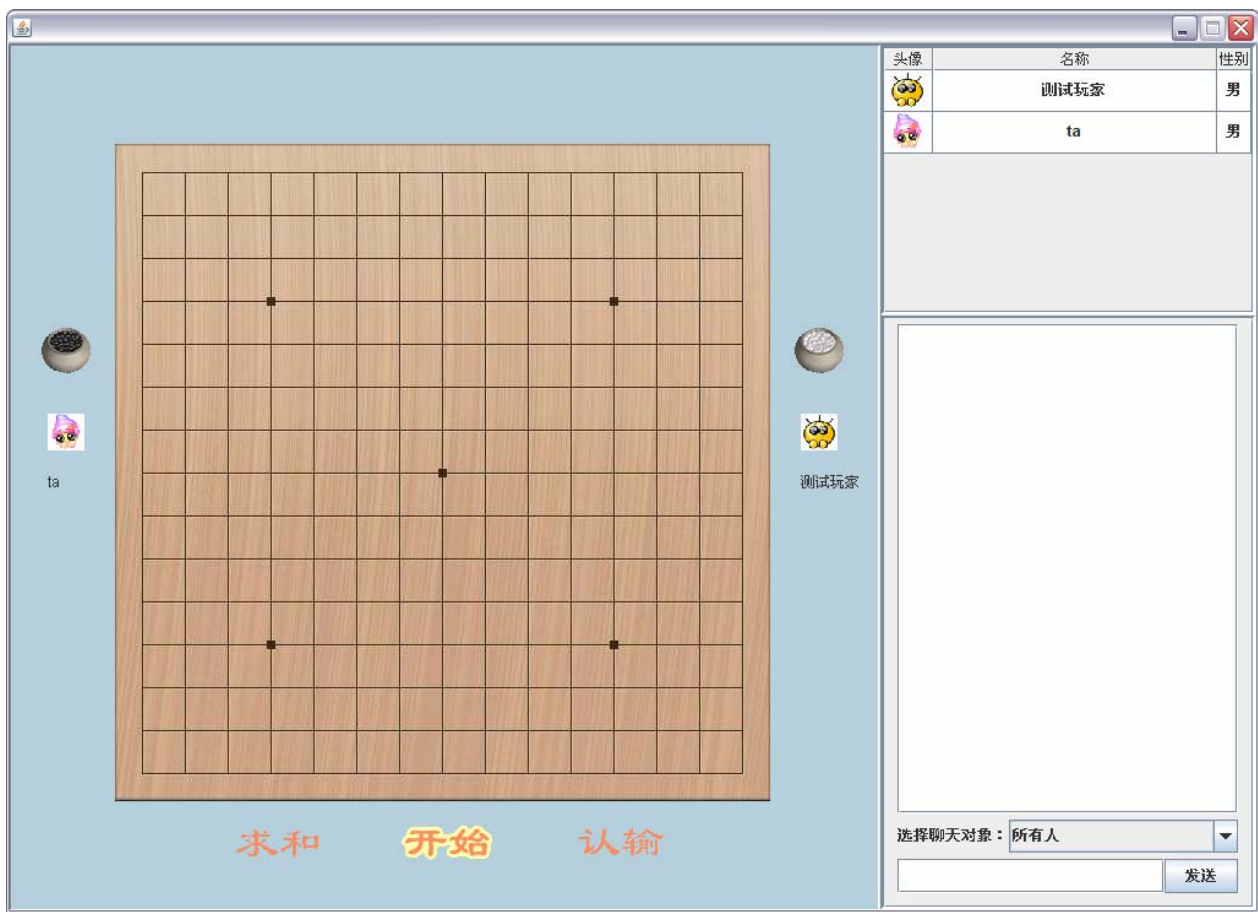


图 15.7 双方玩家坐到桌子上

### 15.5.3 实现游戏聊天

我们已经在 15.4 中实现了游戏大厅的聊天功能，在游戏界面进行聊天时，使用的是同一个界面组件（ChatPanel），因此在游戏中聊天，只需要重新创建一个 ChatPanel 对象即可，但是在创建该对象时，需要提供不同的聊天服务器处理类和客户端处理类。在游戏中聊天的服务器处理类是 GameMessageAction，客户端处理类是 ReceiveMessageAction。

代码清单：

code\GameHall-Server\src\org\crazyit\gamehall\server\action\GameMessageAction.java

```
String senderId = (String)request.getParameter("senderId");
ChessUser sender = ChessContext.users.get(senderId);
String content = (String)request.getParameter("content");
//得到发送人所在的桌子
Table table = ChessContext.getTable(senderId);
if (table != null ) {
    //向对手发送
    ChessUser receiver = table.getAnotherUser(sender);
    if (receiver != null) {
        response.setData("content", sender.getName() + " 对你说: " + content);
        receiver.getPrintStream().println(XStreamUtil.toXML(response));
    }
}
```



`GameMessageAction` 接收到玩家发送的聊天信息后，就向它的对手发送聊天信息，所作的处理与游戏大厅中的聊天类似。`ReceiveMessageAction` 与游戏大厅中所作的处理一样，都是向 `ChatPanel` 中的文本域追加文本。但是需要注意的是，由于我们使用了一个 `UIContext` 中的一个 `Map` 对象来保存界面组件，因此虽然重用了 `ChatPanel`，但是它们是两个实例，因此需要为这两个实例提供不同的名称。才可以使用 `UIContext` 来获得相应的实例。

## 15.6 开始游戏

当双方玩家坐到同一张桌子上，并且都点击了工具栏中的开始时，游戏就正式开始，在本章中，我们确定左边位置的玩家使用黑棋，并可以先下棋，右边的玩家使用白棋。

### 15.6.1 游戏准备

一方玩家坐到桌子上的时候，工具栏中只可以点击开始，当玩家点击了开始的时候，该玩家就处于准备状态，`ChessUser` 中提供了一个 `ready` 的布尔值来表示该玩家是否准备中的状态。玩家点击了开始，除了需要改变他的状态外，还需要告诉对手，他已经准备好了，如果对手都已经准备好了，那么游戏正式开始。为了表示游戏已经开始，我们在游戏区（`GamePanel`）中保存一个布尔值，表示当前游戏的状态。如果玩家已经处理游戏准备的状态，那么就需要更换工具栏图片，如果已经在游戏状态，同样地也需要更换另外的工具栏图片。

图 15.8 是开始游戏的工具栏。



图 15.8 开始游戏的工具栏图片

图 15.9 是准备游戏的工具栏图片



图 15.9 准备游戏的工具栏图片

图 15.10 是游戏中的工具栏图片



图 15.10 游戏中的工具栏图片

当玩家点击了开始时，就需要向服务器发送准备的请求，以下是 `GamePanel` 中的 `ready` 方法。代码清单：

code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\game\GamePanel.java

```
//玩家准备游戏
private void ready() {
    if (this.user.isReady()) return;
    //设置玩家的状态
```

```

        this.user.setReady(true);
        this.currentToolImage = tool_ready;
        this.repaint();
        //发送信息给服务器,告诉服务器已经准备好了
        Request request = new Request("org.crazyit.gamehall.fivechess.server.action.ReadyAction",
            "org.crazyit.gamehall.fivechess.client.action.game.StartGameAction");
        request.setParameter("userId", this.user.getId());
        request.setParameter("tableNumber", this.table.getTableNumber());
        //如果对手没有准备,则设置对手的处理类
        request.setParameter("opponentAction",
            "org.crazyit.gamehall.fivechess.client.action.game.OpponentReadyAction");
        this.user.getPrintStream().println(XStreamUtil.toXML(request));
    }

```

以上的 `ready` 方法向服务器发送了一次请求,告诉服务器自己已经准备游戏了,请求参数包括准备游戏的玩家 `id`,桌子编号,服务器处理类是 `ReadyAction`,客户端处理类是 `StartGameAction`,这里需要注意的是,我们向请求参数中加入了一个 `opponentAction` 的客户端处理类,自己准备游戏时,如果对面位置有玩家,并且该玩家同样已经准备开始游戏的话,就使用客户端的 `StartGameAction` 处理类,如果对手没有准备游戏的话,就使用请求参数中的 `OpponentReadyAction` 作为客户端处理类,让服务器告诉对手,自己准备好了。

代码清单: `code\GameHall-Server\src\org\crazyit\gamehall\server\action\ReadyAction.java`

```

public void execute(Request request, Response response, Socket socket) {
    //得到准备游戏的玩家
    String userId = (String)request.getParameter("userId");
    //得到桌子编号
    Integer tableNumber = (Integer)request.getParameter("tableNumber");
    //得到玩家
    ChessUser user = ChessContext.users.get(userId);
    user.setReady(true);
    //判断对方是否已经准备游戏
    //得到桌子对象
    Table table = Table.getTable(tableNumber, ChessContext.tables);
    Seat seat = table.getUserSeat(user);
    //得到对手
    ChessUser opponent = table.getAnotherSeat(seat).getUser();
    if (opponent != null) {
        //对面座位有人,再判断对手是否已经准备好了
        if (opponent.isReady()) {
            //创建棋盘数组
            createChessArray(table);
            //向双方玩家发送响应,游戏开始
            opponent.getPrintStream().println(XStreamUtil.toXML(response));
            user.getPrintStream().println(XStreamUtil.toXML(response));
        }
        //告诉对手自己准备好了,使用对手接收准备的客户端处理类
        String opponentAction = (String)request.getParameter("opponentAction");
        response.setActionClass(opponentAction);
        response.setData("userId", userId);
        opponent.getPrintStream().println(XStreamUtil.toXML(response));
    }
}

```

ReadyAction 中得到桌子与游戏准备的玩家后，再告诉对手，当前的玩家已经准备好了，如果双方都已经准备好了，那么就创建棋盘的二维数组，棋盘二维数组里面存放的是 Chess 对象，在本章中，一个棋子使用一个 Chess 对象来表示，Chess 对象放在 fivechess-commons 模块，Chess 对象的属性如下：

```
private int beginX; //棋子的开始 X 坐标
private int beginY; //棋子的开始 Y 坐标
private int i; //在二维数组中的一维值
private int j; //在二维数组中的二维值
private String color; //棋子颜色
private Rectangle range; //该棋子的区域
```

由于 Chess 对象是客户端与服务器端共同使用的类，因此可以将这个对象放到 fivechess-commons 模块中。服务器创建了数组后，就可以告诉双方游戏开始。在创建棋盘数组时，我们需要将创建的棋盘数组放到 ChessContext 中，也就是将棋盘的二维数组放到服务器中进行保存，保存的数据结构是 Map，这个 Map 的 key 是桌子编号，value 是 Chess 的二维数组：Map<Integer, Chess[][]>。需要注意的是，玩家在下棋的时候，服务器再对这个二维数组进行遍历，判断输赢，因此判断输赢的操作由服务器进行，并不由客户端进行输赢的判断。OpponentReadyAction 接收到对方准备游戏的服务器响应后，只是设置对手的状态，再对界面组件进行一次 repaint。如果双方都已经准备了游戏，那么服务器就需要发送信息给双方，让两边的客户端去创建棋盘的二维数组，最后调用 GamePanel 的 startGame 方法来开始游戏。

以下是 GamePanel 的 startGame 方法。

代码清单：

code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\game\GamePanel.java

```
//设置游戏状态
public void startGame() {
    this.gaming = true;
    this.currentToolImage = tool_drawAndLost;
    //设置开始游戏的提供
    if (getUserSide().equals(Seat.LEFT)) { //自己先下棋
        this.myTurn = true;
        this.gameStartImage = ImageUtil.getImage("images/fivechess/start-game-you-first.gif");
    } else { //对手先下棋
        this.gameStartImage = ImageUtil.getImage("images/fivechess/start-game-opponent-first.gif");
    }
    this.selectImage = getSelectImage();
    this.startGameTask = new StartGameTask(this);
    this.timer = new Timer();
    timer.schedule(this.startGameTask, 0, 20);
}
```

开始游戏需要设置 gaming 为 true，表示当前正在游戏中，在 GamePanel 中还提供一个 myTure 的布尔值，标识是否轮到当前玩家下棋，在 startGame 方法的最后，启动一个 Timer，将开始游戏的几个字作出动画的效果。开始游戏的具体效果如图 15.11 所示。

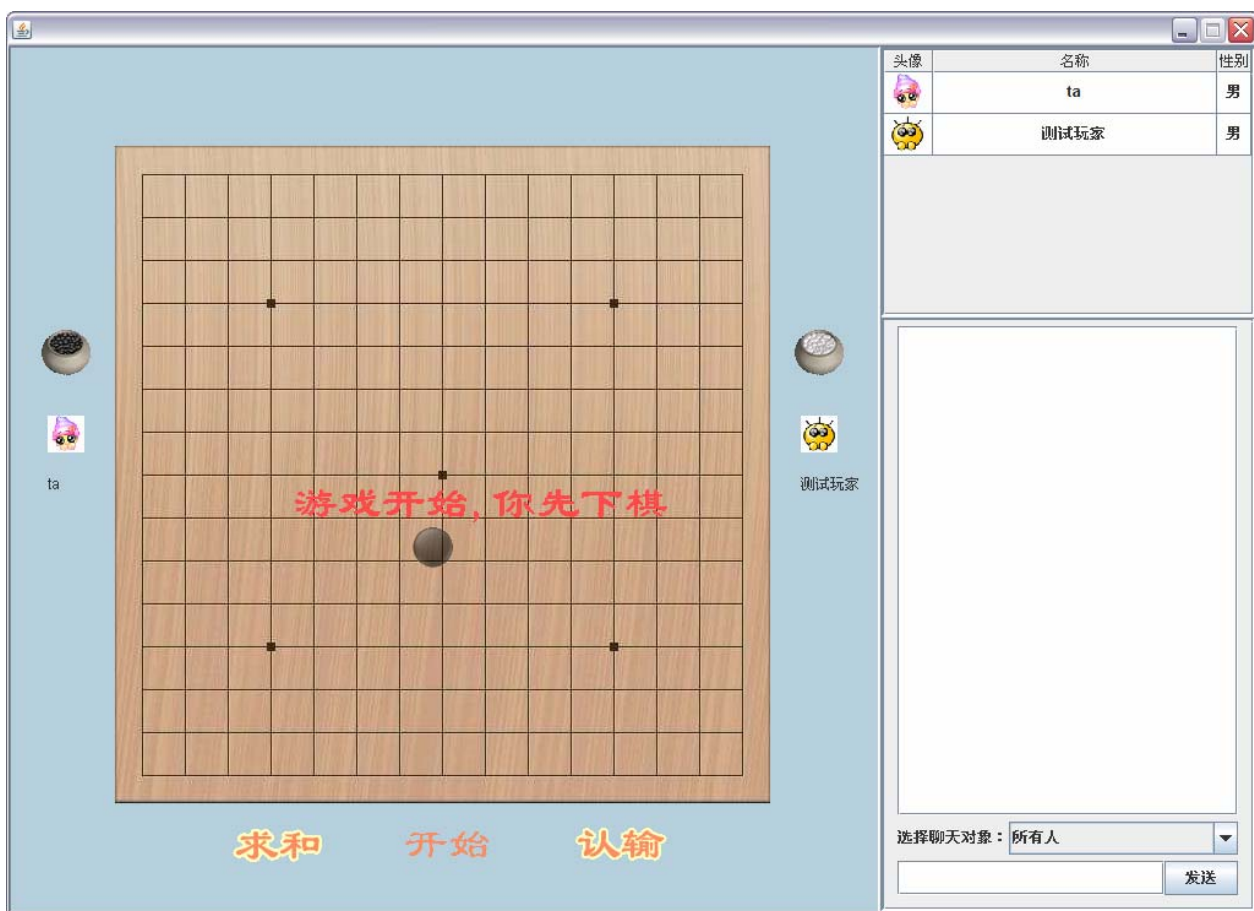


图 15.11 游戏开始

图 15.11 中的“游戏开始, 你先下棋”几个字会慢慢向下, 最后消失, 提示玩家游戏开始, 并且说明下棋的顺序。那么白棋一方看到的效果就是“游戏开始, 对方先下棋”。在开始游戏的时候, 就需要为客户端创建棋盘的二维数组。客户端创建棋盘二维数组的方法与服务器端创建的方式一样。这里需要注意的是, 下棋的一方, 鼠标移动时, 会有一张准备下棋的图片跟随着鼠标的光标, 只需要在鼠标移动时加入选择的图片并在 `paint` 方法中加入判断即可实现。

### 15.6.2 玩家下棋

下棋的玩家在棋盘的某个区域点击了下棋后, 我们界面组件就需要得到鼠标点击的坐标, 再从棋盘二维数组中得到具体的某个 `Chess` 对象。我们棋盘中是一个 `Chess` 的二维数组, 在游戏开始时, 就会初始化这个二维数组 (创建数组中的所有 `Chess` 对象), 该数组中所有的 `Chess` 对象都没有图片, 初始化时只会赋予它们相应的坐标与范围, 当玩家进行了下棋的操作后, 就将玩家对应的棋 (黑棋或者白棋) 的颜色设置到 `Chess` 对象中, `Chess` 对象中有一个 `color` 的属性, 那么在 `GamePanel` 的 `paint` 方法中, 就可以将每一个 `Chess` 都画到界面中。

以下是玩家下棋时所执行的方法:

代码清单:

code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\game\GamePanel.java

//下棋的方法

```
private void takeChess(int x, int y) {  
    Chess chess = getSelectChess(x, y);  
    if (chess != null) {
```

```

        //当前位置有棋子
        if (chess.getColor() != null) {
            UIContext.showMessage("该位置已经有棋子");
        } else {
            //设置颜色
            chess.setColor(getChessColor());
            //轮到对方下棋
            this.myTurn = false;
            //设置选择图片为空
            this.selectImage = null;
            //向服务器发送请求
            requestTakeChess(chess);
            this.repaint();
        }
    }
}

```

玩家下完棋，还需要发送一次请求到服务器，告诉服务器下棋了，服务器需要做的是将玩家下棋的信息保存到服务器的二维数组中并判断是否胜利，如果没有胜利，还需要将玩家的下棋信息发送给对手，让对手去更新他自己的界面组件并且轮到对手下棋。这里需要考虑的是请求中应该存放一些什么样的参数给服务器，由于玩家在某个位置下棋，该位置对应的是某个 **Chess** 对象，因此我们可以将这个 **Chess** 对象在二维数组中的一维值（i）与二维值（j）作为请求参数，另外还要告诉服务器是哪个玩家在哪个桌子中下的棋，并且下棋的颜色是什么，除了这些信息外，如果玩家的这一步棋影响了游戏的结果（胜利了），就需要让对手去处理失败的动作。

```

//告诉服务器自己下棋了
private void requestTakeChess(Chess chess) {
    Request request = new Request(
        "org.crazyit.gamehall.fivechess.server.action.TakeChessAction",
        "org.crazyit.gamehall.fivechess.client.action.game.TakeChessAction");
    //设置请求的各个参数
    request.setParameter("i", chess.getI());
    request.setParameter("j", chess.getJ());
    request.setParameter("userId", this.user.getId());
    request.setParameter("tableNumber", this.table.getTableNumber());
    request.setParameter("color", chess.getColor());
    //设置处理胜利的 Action
    request.setParameter("winAction",
        "org.crazyit.gamehall.fivechess.client.action.game.WinAction");
    //设置处理输的 Action
    request.setParameter("lostAction",
        "org.crazyit.gamehall.fivechess.client.action.game.LostAction");
    this.user.getPrintStream().println(XStreamUtil.toXML(request));
}

```

需要注意的是，以上的代码设置了处理胜利的 **Action** 和处理失败的 **Action**，如果当前玩家下的棋导致玩家游戏胜利的话，那么就是当前的玩家执行 **WinAction**，由于自己胜利了，对手自然就是失败，对手执行 **LostAction**。如果没有胜利的话，对手就执行客户端处理类 **TakeChessAction**。

服务器端的 **TakeChessAction** 需要将当前玩家下的棋子信息保存到服务器端的二维数组，再对该数组进行遍历，判断是否胜利。

以下是服务器端的 **TakeChessAction** 判断是否胜利的代码。

代码清单：code\GameHall-Server\src\org\crazyit\gamehall\server\action\TakeChessAction.java

```

//判断是否胜利
boolean win = validateWin(chessArray, chessArray[i][j]);
if (opponent == null) win = true;
//告诉双方，赢了
if (win) {
    //告诉赢的一方
    tellWin(request, user, response);
    //告诉输的一方
    tellLost(request, opponent, response);
    //设置服务器中双方的状态
    opponent.setReady(false);
    user.setReady(false);
}

```

以上的代码中使用了一个 `win` 的布尔值，该值取决于 `validateWin` 方法，`validateWin` 方法用于遍历服务器的二维数组，遍历一个二维数组中的每一个 `Chess` 对象，判断是否可以横、竖、斜连成五个棋子。遍历的时候，需要纵向遍历、横向遍历、从上往下斜向遍历、从下往上斜向遍历。以下是纵向遍历的代码。

代码清单：code\GameHall-Server\src\org\crazyit\gamehall\server\action\TakeChessAction.java

```

//纵向遍历
private boolean vertical(Chess[][] chessArray, Chess chess) {
    //连续棋子的总数
    int count = 0;
    for (int i = 0; i < chessArray.length; i++) {
        if (i == chess.getI()) {
            for (int j = 0; j < chessArray[i].length; j++) {
                Chess c = chessArray[i][j];
                if (c.getColor() != null && c.getColor().equals(chess.getColor())) {
                    count++;
                }
            }
        }
    }
    if (count >= 5) return true;
    return false;
}

```

如果 `Chess` 对象的颜色与玩家刚才所下的棋子颜色一致并且可以连成五个，那么就返回 `true`，表示下的该棋子导致游戏结束（下棋的玩家胜利了）。玩家胜利后，就调用 `tellWin` 方法告诉下棋的一方：你赢了，调用 `tellLost` 方法告诉对方：你输了。赢的一方就会执行客户端处理类 `WinAction`，输的一方就会执行客户端处理类 `LostAction`。`WinAction` 与 `LostAction` 都是将游戏重新进行初始化，再告诉玩家游戏结果。游戏初始化需要进行如下操作：

- ❑ 将下棋时候的选择图片设置为 `null`
- ❑ 将游戏状态（`GamePanel` 的 `gaming` 属性）设置为 `false`
- ❑ 将轮到自己下棋的标识（`GamePanel` 的 `myTurn` 属性）设置为 `false`
- ❑ 将客户端双方玩家的准备标识（`ChessUser` 的 `ready` 属性）设置为 `false`
- ❑ 设置工具栏图片为图 15.8（只允许点击开始）

玩家下棋的代码已经实现，具体效果如图 15.12 所示。



图 15.12 下棋游戏胜利

### 15.6.3 逃跑与认输

一方玩家由于特殊原因，点击了认输或者直接关掉游戏，那么服务器就直接判定该玩家输，并提示对方胜利，认输与关掉游戏，都可以发送一次请求到服务器，让服务器进行相关的处理。

以下是 **GamePanel** 的认输方法。

代码清单：

code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\game\GamePanel.java

```
//发送认输的请求
public void sendLostRequest() {
    Request request = new Request(
        "org.crazyit.gamehall.fivechess.server.action.LostAction",
        "org.crazyit.gamehall.fivechess.client.action.game.OpponentLostAction");
    request.setParameter("userId", this.user.getId());
    request.setParameter("tableNumber", this.table.getTableNumber());
    this.user.getPrintStream().println(XStreamUtil.toXML(request));
}
```

以上代码构造一次请求发送到服务器，服务器的处理类是 **LostAction**，客户端处理类是 **OpponentLostAction**，客户端处理类是由赢的一方（认输方的对手）执行的。客户端处理类得到认输的请求后，就将该玩家的所有状态（服务器中的状态）都还原，发送信息给对手：你赢了。请求认输我们需要将认输玩家的 id，桌子编号等信息告诉服务器，因此需要添加相关的请求参数。

处理逃跑的 **Action** 与认输一样，只是当用户关闭游戏窗口时执行发送请求，服务器端与客户端处理

类执行的操作与认输类似。

### 15.6.4 请求和棋

请求和棋的发送方向服务器发送一次请求，让服务器找到他的对手，向对手询问：是否同意和棋。如果对手同意和棋，那么就发送信息到服务器，告诉服务器他们和棋了，服务再向双方发送和棋的信息，让双方显示和棋的提示。如果对手拒绝和棋，同样地，也会发送一次请求到服务器，让服务器转发，告诉和棋请求人，对方拒绝和棋。到目前为止，游戏大厅中所有的相关请求都通过 **Request** 来发送信息到服务器，告诉服务器应该作如何处理，服务器再找到相关的客户端并发送相应的服务器响应 (**Response**)。

以下是 **GamePanel** 中请求和棋的方法。

代码清单：

code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\game\GamePanel.java

```
//请求和
private void requestDraw() {
    //询问是否求和
    int result = UIContext.showConfirm("你确定要求和吗?");
    if (result == 0) {
        //向服务器发送求和请求
        Request request = new Request(
            "org.crazyit.gamehall.fivechess.server.action.DrawAction",
            "org.crazyit.gamehall.fivechess.client.action.game.DrawAction");
        request.setParameter("userId", this.user.getId());
        request.setParameter("tableNumber", this.table.getTableNumber());
        this.user.getPrintStream().println(XStreamUtil.toXML(request));
    } else {
        return;
    }
}
```

以上代码构造一个 **Request** 对象向服务器发送求和请求，需要将请求人的 **id**（玩家 **id**）与桌子编号设置到参数中，那么服务器就可以知道哪张桌子的哪个玩家请求和棋，就可以根据这些信息得到发送和棋请求的玩家的对手，再向该对手发送和棋询问。和棋的服务器处理类是 **DrawAction**，客户端处理类是 **DrawAction**，这两个 **Action** 名字相同，但是存在的模块不一样。本章中所有的服务器处理类都存在于 **fivechess-server** 模块，所有的客户端处理类都存在于 **fivechess-client** 模块。

以下是 **GamePanel** 中同意求和的方法。

代码清单：

code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\game\GamePanel.java

```
//同意求和
private void agreeDraw() {
    draw();
    //告诉服务器同意求和
    Request request = new Request("org.crazyit.gamehall.fivechess.server.action.AgreeDrawAction",
        "org.crazyit.gamehall.fivechess.client.action.game.AgreeDrawAction");
    request.setParameter("userId", this.user.getId());
    request.setParameter("tableNumber", this.table.getTableNumber());
    this.user.getPrintStream().println(XStreamUtil.toXML(request));
}
```

以上的黑体代码调用了 **draw** 方法，该方法只是普通的弹出提示与初始化游戏，提示和棋后，再发送一次请求到服务器，告诉服务器同意求和。服务器处理类是 **AgreeDrawAction**，客户端处理类（发送



求和请求一方)是 **AgreeDrawAction**。当服务器端接收到同意求和的请求后,就可以初始化游戏双方在服务器中的状态,包括设置玩家(**ChessUser**)的 **ready** 属性,再告诉发送求和信息的一方,对方同意了和棋,发送求和信息的一方得到服务器的响应后,同样的把发送求和信息一方的所有状态都改变成为初始化状态,可以继续下一局的游戏。

以下是 **GamePanel** 中拒绝求和的方法。

代码清单:

code\fivechess-client\src\org\crazyit\gamehall\fivechess\client\ui\game\GamePanel.java

```
//拒绝求和
private void refuseDraw() {
    //告诉服务器拒绝和棋
    Request request = new Request("org.crazyit.gamehall.fivechess.server.action.RefuseDrawAction",
        "org.crazyit.gamehall.fivechess.client.action.game.RefuseDrawAction");
    request.setParameter("userId", this.user.getId());
    request.setParameter("tableNumber", this.table.getTableNumber());
    this.user.getPrintStream().println(XStreamUtil.toXML(request));
}
```

同样地,拒绝求和也是构造一次请求发送到服务器,让服务器去告诉发送求和信息的一方:对方拒绝了你的求和要求。发送的一方得到服务器响应后,就执行客户端处理类 **RefuseDrawAction**,提示相关的信息。

## 15.7 五子棋游戏大厅总结

到此,五子棋游戏大厅已经全部完成,可能还有一些细节可以做得更好,例如提示信息不必使用弹出提示等。五子棋游戏大厅使用了我们在 15.2 中编写的游戏大厅框架来实现,所有的客户端与服务器通信,都是使用 **Request** 对象,服务器使用 **Response** 对象作出响应,所有的请求参数与响应参数都设置到这两个对象中,我们在开发五子棋游戏大厅的时候,基本上没有接触过任何的 **Socket** 对象,只需要构造一次请示,将服务器处理类与客户端处理类放到请求中,让我们的服务器去帮助我们寻找服务器处理类,让客户端去寻找客户端处理类。在本章中,框架的服务器端是 **fivechess-server** 模块,框架的客户端是 **fivechess-client** 模块。如果我们编写了新的游戏,就可以直接将游戏的服务器包放到 **fivechess-server** 中,将游戏的客户端的包放到 **fivechess-client** 中,编写游戏的时候,并不需要关心如何进行通信,只需要关心具体的业务实现,前提是必须遵守这个游戏大厅框架所定的一些规则。

在本章的案例中,源文件所存放的是 **gamehall-src** 目录,游戏运行的目录是 **gamehall-run**,该目录下存在有 **crazyit-gamehall-client** 与 **crazyit-gamehall-server** 目录,读者在运行游戏的时候,先打开 **crazyit-gamehall-server/bin** 下的 **startup.bat**,就可以启动服务器,运行客户端的话,打开 **crazyit-gamehall-client/bin** 下的 **startup.bat** 就可以启动客户端。以下是服务器端与客户端的目录说明。

**crazyit-gamehall-server:**

- ❑ **bin** 目录: 包含一些启动服务器的命令;
- ❑ **game** 目录: 存放游戏服务器端的 jar 包;
- ❑ **lib** 目录: 存放一些服务器端所必需的包,例如 **gamehall-server** 模块的包、**xstream** 的包等。

**crazyit-gamehall-client:**

- ❑ **bin** 目录: 包括一些启动客户的命令;
- ❑ **game** 目录: 存放游戏客户的 jar 包;
- ❑ **images** 目录: 各个游戏所需要的图片存放目录;
- ❑ **lib** 目录: 存放一些客户端所必需的包。

## 15.8 编写一个测试聊天室

我们编写了一个游戏大厅的框，我们再次编写一个简单的聊天室，并将这个聊天室的客户端与服务端打成 jar 包，分别放到框架的服务器目录与客户端目录，测试我们的框架是否支持游戏的扩展。

### 15.8.1 建立聊天室界面

聊天室界面十分简单，一个 JTextArea、JTextField、发送消息的按钮与用户列表即可。具体效果如图 15.13 所示。

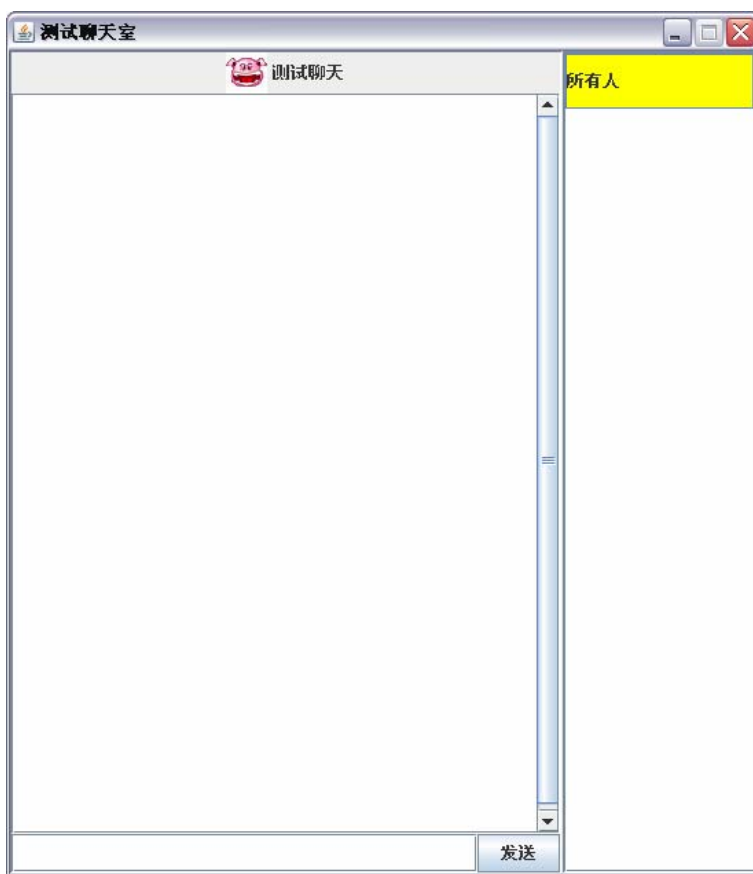


图 15.13 聊天室界面

### 15.8.2 实现聊天室

聊天室所包含的动作有：用户进入聊天室、发送聊天信息、接收聊天信息等。我们在聊天室的服务器端模块中加入一个聊天室上下文，用于保存进入聊天室的全部用户。当有新的用户进入聊天室时，我们就向其他在线的用户发送信息。这样的一个过程，在我们编写的框架中，只需要构造一个 Request 对象并发送给服务器即可实现，服务器再作出相关的响应，在 JTextArea 中显示相关的信息。

开发客户端处理类与五子棋一样，都是通过一个界面的上下文类来得到界面组件对象，服务器作出响应时，就会执行这些客户端处理类，然后再对界面组件进行相关的控制。

聊天室的具体效果如图 15.14 所示。



图 15.14 聊天室完成效果

基于我们所编写的框架来完成一个聊天室十分简单，只需要简单的界面代码和几个处理类即可，可以看到我们所编写的框架具有扩展性，如果需要编写其他游戏加入到该框架中，只需要编写相关的 **Action** 类即可，我们可以不用去关心如何进行信息传输。由于聊天室的实现相对较为简单，只需要遵守框架的几个规则即可，因此具体的编写代码在此不详细描述，可以具体参看源代码中的 `chat-room-client` 模块与 `chat-room-server` 模块。

### 15.8.3 将聊天室放置到框架中测试

我们的框架中有一个最基本的规则，就是需要在客户端的包的 `MANIFEST.MF` 中加入 `Game-Class` 属性，来声明客户端的游戏入口类，因此聊天室的 `MANIFEST.MF` 文件内容如下。

代码清单：`code\chat-room-client\src\META-INF\MANIFEST.MF`

```
Manifest-Version: 1.0
```

```
Game-Class: org.crazyit.gamehall.chatroom.client.ChatIndex
```

聊天室的 `MANIFEST.MF` 文件声明了由 `ChatIndex` 类作为聊天室的入口类，当 `GameHall-Client` 模块启动登录界面的时候，就会到客户端的 `game` 目录去加载所有的游戏客户端的包，并读取这些包中的 `MANIFEST.MF` 文件中的 `Game-Class` 属性，这样才会向登录界面的游戏下拉框架中加入一个新的游戏，登录界面的具体效果如图 15.15 所示。



图 15.15 登录界面选择游戏

这样，我们的框架就可以支持游戏的加入删除，只需要在 `crazyit-gamehall-client` 的 `game` 目录下加入一个新的 `jar` 包就可以实现加入游戏。

## 15.9 本章小节

本章开发了一个游戏大厅的框架，该框架主要用于处理服务器与客户端之间的信息传输，并且在些基础上开发了一个五子棋的游戏大厅与一个简单的聊天室。讲解 `Socket` 编程的相关知识点。在开发五子棋游戏大厅时，主要讲解了五子棋游戏大厅的实现原理。本章的重点是游戏大厅框架的开发，让我们的这个游戏大厅框架可以做到动态的加载游戏，让玩家选择进入的游戏，当加入其他游戏时，该框架并不需要对原来的代码进行修改，按照一定的规则就可以加入新的游戏。本章开发的这个游戏大厅框架，希望能给读者带来一些编程上的启发，开发出更多优秀的基于网络的游戏。